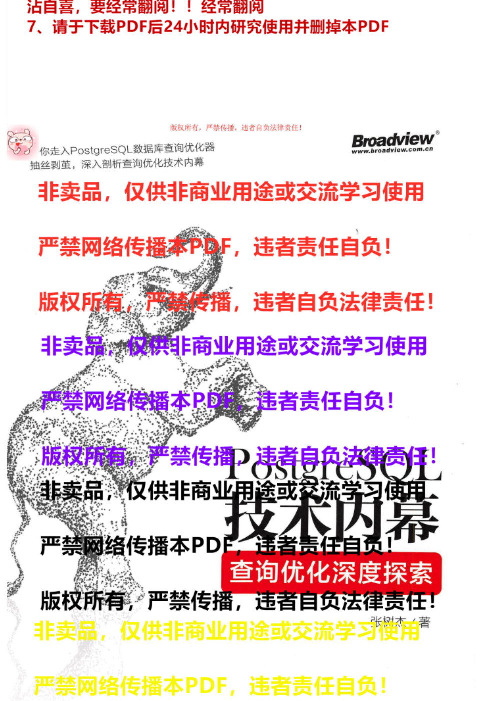


版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权所有，严禁传播，违者自负法律责任！



非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！
版权所有，严禁传播，违者自负法律责任！



张树杰

拥有13年IT从业经验，从事国产数据库内核开发工作多年，对数据库内核各个方面均有涉猎，近些年专注于研究如何改进分布式数据库的查询优化、查询执行，目前在Pivotal公司从事Apache HAWQ数据库的内核开发工作。

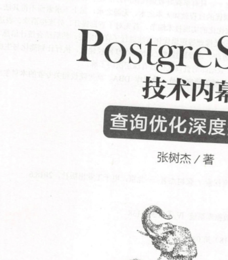
邮箱：zhangshujie@pku.edu.cn

非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！
版权所有，严禁传播，违者自负法律责任！

PostgreSQL 技术内幕

查询优化深度探索

张树杰 / 著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！
版权所有，严禁传播，违者自负法律责任！

内 容 简 介

查询优化器是数据库中很重要的模块之一，只有掌握好查询优化的方法才能了解查询优化的细节，在对数据库调优的过程中才能有的放矢，否则调优的过程就如无本之木、无源之水。[illegible]

本书基于了PostgreSQL数据库中查询优化的实现[illegible]式简化理，详细下推、连接顺序交换、等价类推理等[illegible]选择率、代价计算以及路径生成[illegible]等进行了深度探索，使读者对PostgreSQL数据库的查询优化[illegible]

本书适合数据库内核开发人员[illegible]的研究人员，数据库DBA、高等院校相关专业的本科生或研究生阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

PostgreSQL技术内幕：查询优化深度探索 / 张树杰著. —北京：电子工业出版社，2018.6
ISBN 978-7-121-34148-9

Ⅰ. ①P… Ⅱ. ①张… Ⅲ. ①关系数据库系统 Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字（2018）第088240号

责任编辑：董英
印　　刷：三河市良远印务有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
北京市海淀区万寿路173信箱　邮编：100036
开　　本：787×980　1/16　印张：21.75　字数：558千字
版　　次：2018年6月第1版
印　　次：2018年6月第1次印刷
定　　价：79.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。
质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。
本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！
版权所有，严禁传播，违者自负法律责任！

序一

查询优化是数据库管理系统中最重要的功能之一。每一个生产数据库系统均[illegible]

作为最先进的开源对象关系数据库系统，PostgreSQL及其源代码[illegible]

身为一名PostgreSQL爱好者和数据库研究人员，[illegible]

彭煜玮
2018.4.25 于珞珈山

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权所有，严禁传播，违者自负法律责任！

序二

中国有句古话，“巧妇难为无米之炊”，说的是再好的主妇，在没有给任何食材的情况下也做不出可口的饭菜。反过来，什么样的主妇算得上“巧妇”呢？如果给你准备好了烹调所需的所有食材，你能做出可口的饭菜吗？

数据库是一个比较神奇的软件，我们都知道可以用SQL和数据库沟通，让数据库处理SQL和让主妇做饭是一样的道理，数据库能不能及时响应SQL请求，能不能用最优的计划完成SQL请求，取决于数据库本身提供了哪些“料”，以及数据库打算怎么“烹调”用户提交的SQL。例如，一个简单查询SQL，数据库的扫描方法（我暂且把它称为数据库的“料”之一）就有全表扫描、索引扫描、位图扫描、跳跃扫描等。一个SQL中包括了多个函数、表达式时，数据库先处理哪个表达式或函数，又或者在什么时候处理这些表达式或函数呢？数据库“烹调”一条SQL时，如何“烹调”，如何分解，是靠什么来做决定的？在数据库决定了怎么做之后，又是如何按部就班地执行的？如果说数据库的“扫描方法、表达式、操作符、UDF、索引接口”等是数据库的“料”，那么数据库的优化器就是“巧妇”之手，它包括了“JOIN算法、SQL重写规则、多表JOIN的遗传算法、动态路径规划、选择性算法、各种NODE的成本计算算法、成本因子、并行计算成本算法”等方方面面，为数据库如何执行SQL提供了全套流程。

PostgreSQL作为一个非常经典的ORDBMS，包含了很多“料”，同时有着非常先进的优化器，为高效地执行SQL提供了良好的基础。

本书作者长期致力于数据库内核的研发，有非常丰富的理论与实践经验，本书以PostgreSQL为背景，详细介绍了PostgreSQL查询优化器中的核心概念，从“查询树、SQL重写、UNION优化、逻辑分解”到“下推、JOIN、选择性、统计信息、扫描路径、动态规划、遗传算法”等方方面面，实为作者呕心沥血之作，同时也是数据库工作者，特别是PGer之福。

非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！
版权所有，严禁传播，违者自负法律责任！

本书是不可多得的教科书级PostgreSQL内核读物，同时不乏实战性。建议想了解数据库优化器工作原理的读者及PostgreSQL爱好者深入学习。

感谢作者为PostgreSQL生态的辛勤付出，期待本书大卖。

PGer，Digoal

非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！
版权所有，严禁传播，违者自负法律责任！

前　言

为什么写这本书

我参加过很多次查询优化的培训，也查阅过很多查询优化的资料，但总是感觉对查询优化似懂而非，我总结其原因是多数培训和资料的时长或篇幅较短，内容多是对查询优化的概述，“巧妙”地避开了查询优化的难点，难以触及查询优化的本质，导致查询优化的“大道理”人人都懂，遇到问题却难以发力。

2016年年末，我做了一次查询优化的培训，结合之前培训的经验，我对这次查询优化的培训打了一个“持久战”，不只是拿出几个小时的时间对查询优化进行一个总体描述，而是将查询优化器拆解开来，分阶段地进行详细的解读，大约做了十几次培训，最终的效果是非常显著的。在培训的过程中我发现，目前PostgreSQL数据库查询优化器实现细节相关的资料市场上少之又少，和数据库从业人员对查询优化器的热情远远不成正比，本着抛砖引玉的原则，我写了这本书。

为什么阅读这本书

- 在数据库内核开发的过程中，你是否有了解查询优化器的实现细节的欲望？
- 在对数据库进行调优的过程中，你是否感觉无从下手？

非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！
版权所有，严禁传播，违者自负法律责任！

- 在分析查询优化的源码时，你是否会陷入某一细节而不可自拔？
- 在学习查询优化的理论时，你是否感觉理论与实践之间无法一一对应？

如果你希望深入地了解查询优化，那么最好的办法就是了解它的理论基础，然后细致地剖析查询优化器的源代码，通过理论和实践的结合，达到真正掌握相关知识的目的。本书细致地解读了PostgreSQL 10.0的查询优化器的大部分源码，对其中比较重要的理论都给出了说明，足以让读者了解PostgreSQL数据库查询优化器的全貌。

虽然本书已经尽量尝试将复杂问题简单化，但是鉴于PostgreSQL数据库的查询优化器的实现本身就具有一定的复杂性，读者阅读的过程可能是“痛苦”的，但请相信“梅花香自苦寒来”，只要坚持阅读就能收获很多。

本书的组织结构

本书的组织结构基本是按照PostgreSQL数据库的查询优化器处理一个查询的流程来安排的，由简入繁、由易入难。

第1章介绍一些查询优化基础理论，这些理论是对查询优化的概述，读者在阅读第1章时可以参考一些经典的数据库实现理论书籍，更详细地了解数据库的基本理论，这样能给后面的阅读打好基础。

第2章介绍查询树，查询树是PostgreSQL数据库查询优化器的输入，查询优化器本身是对查询树的等价改造及等价分解。

第3章介绍逻辑重写优化，逻辑重写优化是逻辑优化的一部分，它主要是对查询树进行基于规则的等价重写，比较重要的有子查询提升、表达式预处理、外连接消除等。

第4章介绍逻辑分解优化，逻辑分解优化仍然是逻辑优化的一部分，和逻辑重写优化不同，它开始尝试分解查询树，经过谓词下推、连接顺序交换、等价类推理等对查询树进行改造。

第5章介绍统计信息和选择率，统计信息是代价计算的基石，因此了解统计信息的类型、了解选择率的含义对代价计算有非常重要的意义。

第6章介绍扫描路径的建立过程，扫描路径是为了对基表进行扫描的物理算子创建的路径，它负责将物理存储或者缓存中的数据读取上来并进行处理，通常包括顺序扫描、索引扫描、位图扫描等。

第7章介绍路径搜索的两个算法，PostgreSQL数据库采用了动态规划方法和遗传算法进行

非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！
版权所有，严禁传播，违者自负法律责任！

路径搜索，本书对这两种方法的实现都做了详细的介绍。

第8章介绍连接路径的建立过程，PostgreSQL数据库的物理连接路径有嵌套循环连接、哈希连接、归并连接等，由于采用的扫描路径不同，导致同一种类型的物理连接路径产生的代价不同。

第9章介绍Non-SPJ的相关优化，PostgreSQL数据库对集合操作、聚集操作、分组操作、排序操作等都做了优化处理。

第10章介绍执行计划的生成，在扫描路径、连接路径及Non-SPJ路径分别处理之后，会选择一个“最优”的连接树，PostgreSQL数据库需要将这个连接树修正成执行计划树。

错误

限于我的能力，书中难免有错误，在写作的过程中我也尝试尽量多查阅相关的资料，尽量避免错误的出现，但是相关的资料实在是太少了，因此，欢迎广大读者对本书提出纠正、批评和意见，这也有益于我本身能力的提升。

致谢

感谢彭煜玮、周正中（德哥Digoal）为本书作序，感谢蒋志勇、文继军、王颖泽、杨瑜、赵殿奎对本书的评价，这对我是极大的鼓励。

在写作过程中，卢栋栋、彭信东、李茂增通读了大部分书稿，给出了很多有益的意见和建议，在此表示感谢。林文、翁燕青、白洁对书稿的格式及内容提出了修改建议，在此一并表示感谢。

感谢董英编辑，在写稿及后续的审校过程中董英编辑一直在和我沟通，不厌其烦地解答我的各种问题。

感谢我的家人。我的父母和妻子在我写作的过程中给予了极大的支持，写作的过程非常枯燥，他们为我提供了最好的写作环境。另外我的两个儿子也经常在我离开电脑的间隙帮我修改书稿，虽然他们的意见一条也没有被采纳，但这里仍然对他们的“贡献”表示感谢。

严禁网络传播本PDF，违者责任自负！

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权所有，严禁传播，违者自负法律责任！

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/34148

严禁网络传播本PDF，违者责任自负！

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权所有，严禁传播，违者自负法律责任！

目　录

第 1 章　概述 1

1.1　查询优化的简介 1
1.2　逻辑优化 3
1.2.1　关系模型 3
1.2.2　逻辑优化示例 8
1.3　物理优化 10
1.3.1　物理优化的 4 个“法宝” 12
1.3.2　物理路径的生成过程 14
1.4　文件介绍 17
1.5　示例的约定 18
1.6　小结 19

第 2 章　查询树 20

2.1　Node 的结构 20

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

2.2 Var 结构体 21
2.3 RangeTblEntry 结构体 23
2.4 RangeTblRef 结构体 25
2.5 JoinExpr 结构体 26
2.6 FromExpr 结构体 27
2.7 Query 结构体 27
2.8 查询树的展示 31
2.9 查询树的遍历 31
2.10 执行计划的展示 32
2.11 小结 33

第 3 章 逻辑重写优化 34

3.1 通用表达式 35
3.2 子查询提升 36
3.2.1 提升子连接 37
3.2.2 提升子查询 51
3.3 UNION ALL 优化 68
3.4 展开继承表 69
3.5 预处理表达式 71
3.5.1 连接 Var 的溯源 71
3.5.2 常量化简 72
3.5.3 谓词规范 73
3.5.4 子连接处理 79
3.6 处理 HAVING 子句 80
3.7 Group By 键值消除 81
3.8 外连接消除 82
3.9 grouping_planner 的说明 91
3.10 小结 92

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

第 4 章　逻辑分解优化 93

4.1　创建 RelOptInfo 94
4.1.1　RelOptInfo 结构体 94
4.1.2　IndexOptInfo 结构体 97
4.1.3　创建 RelOptInfo 100
4.2　初识等价类 102
4.3　谓词下推 106
4.3.1　连接条件的下推 106
4.3.2　过滤条件的下推 112
4.3.3　连接顺序 113
4.3.4　deconstruct_recurse 函数 118
4.3.5　make_outerjoininfo 函数 124
4.3.6　distribute_qual_to_rels 函数 132
4.3.7　reconsider_outer_join_clauses 函数 151
4.3.8　generate_base_implied_equalities 函数 156
4.3.9　记录表之间的等价关系 157
4.4　PlaceHolderVar 的作用 158
4.5　Lateral 语法的支持 161
4.5.1　Lateral 的语义分析 162
4.5.2　收集 Lateral 变量 164
4.5.3　收集 Lateral 信息 164
4.6　消除无用连接项 166
4.7　Semi Join 消除 171
4.8　提取新的约束条件 172
4.8.1　提取需要满足的条件 173
4.8.2　提取流程 174
4.8.3　选择率修正 176
4.9　小结 177

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

第 5 章　统计信息和选择率 178

5.1　统计信息 178

5.1.1　PG_STATISTIC 系统表 181

5.1.2　PG_STATISTIC_EXT 系统表 185

5.1.3　单列统计信息生成 187

5.1.4　多列统计信息生成 196

5.2　选择率 200

5.2.1　使用函数依赖计算选择率 204

5.2.2　子约束条件的选择率 208

5.2.3　基于范围的约束条件的选择率修正 211

5.3　OpExpr 的选择率 213

5.3.1　eqsel 函数 215

5.3.2　scalargtsel 函数 217

5.3.3　eqjoinsel 函数 220

5.4　小结 226

第 6 章　扫描路径 227

6.1　代价（Cost） 228

6.1.1　代价基准单位 228

6.1.2　启动代价和整体代价 231

6.1.3　表达式代价的计算 233

6.2　路径（Path） 236

6.2.1　Path 结构体 236

6.2.2　并行参数 237

6.2.3　参数化路径 239

6.2.4　PathKey 242

6.3　make_one_rel 函数 244

6.4　普通表的扫描路径 245

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

6.4.1 顺序扫描 246
6.4.2 索引扫描 248
6.4.3 位图扫描 281
6.5 小结 291

第 7 章 动态规划和遗传算法 292

7.1 动态规划 293
7.1.1 make_rel_from_joinlist 函数 297
7.1.2 standard_join_search 函数 298
7.1.3 join_search_one_level 函数 298
7.2 遗传算法 301
7.2.1 种群初始化 303
7.2.2 选择算子 308
7.2.3 交叉算子 310
7.2.4 适应度计算 311
7.3 小结 312

第 8 章 连接路径 313

8.1 检查 314
8.1.1 初步检查 314
8.1.2 精确检查 316
8.1.3 “合法”连接 318
8.2 生成新的 RelOptInfo 324
8.3 虚表 327
8.4 Semi Join 和唯一化路径 328
8.5 建立连接路径 331
8.5.1 sort_inner_and_outer 函数 334
8.5.2 match_unsorted_outer 函数 345

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

8.5.3 hash_inner_and_outer 函数 350
8.6 路径的筛选 355
8.7 小结 360

第 9 章 Non-SPJ 优化 361

9.1 集合操作处理 361
9.2 Non-SPJ 路径 367
9.2.1 Non-SPJ 预处理 368
9.2.2 Non-SPJ 路径生成 376
9.3 小结 382

第 10 章 生成执行计划 383

10.1 转换流程 383
10.1.1 扫描计划 384
10.1.2 连接计划 390
10.2 执行计划树清理 391
10.3 小结 395

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

1 第1章 概述

PostgreSQL 数据库是世界上最先进的开源关系数据库，是数据库从业人员研究数据库的宝贵财富，我们不打算再复述 PostgreSQL 数据库的历史及概况，而是直入主题，看一下世界上最先进的开源数据库中的一个模块——查询优化器的实现方法。

1.1 查询优化的简介

查询优化是数据库管理系统中承上启下的一个模块，如图 1-1 所示，它接收来自语法分析模块传递过来的查询树，在这个查询树的基础上进行了逻辑上的等价变换、物理执行路径的筛选，并且把选择出的最优的执行路径传递给数据库的执行器模块。简而言之，一个查询优化器它的输入是查询树，输出是查询执行计划。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

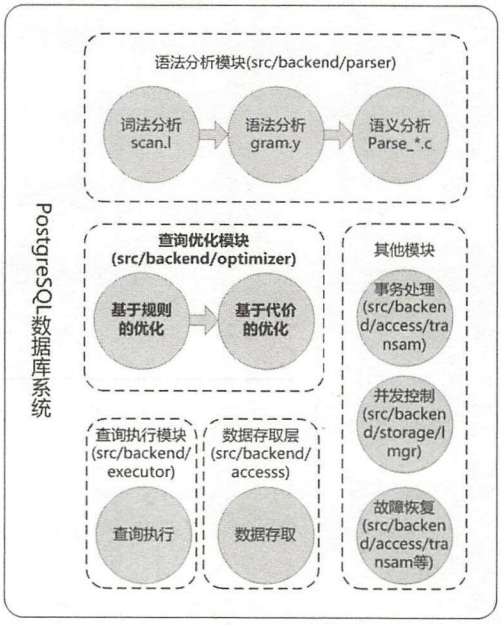


图 1-1　数据库的整体架构

数据库的使用者在书写 SQL 语句的时候也经常会考虑到查询的性能，根据自己已知的情况争取写出性能很高的 SQL 语句，但是一个应用程序可能要写大量的 SQL 语句，而且有些 SQL 语句的逻辑极为复杂，数据库应用开发人员很难面面俱到地写出“极好的”语句，而查询优化器相对于数据库应用开发人员而言，具有一些独特的优势：

- 查询优化器和数据库用户之间的信息不对称，查询优化器在优化的过程中会参考数据库统计模块自动产生的统计信息，这些统计信息从各个角度来描述数据的分布情况，查询优化器会综合考虑统计信息中的各种数据从而得到一个好的执行方案，而数据库用户一方面无法全面地了解数据的分布情况，另一方面即使数据库用户获得了所有的统计数据，人脑也很难构建一个精确的代价计算模型来对执行方案进行筛选。
- 查询优化器和数据库用户之间的时效性不同，数据库中的数据瞬息万变，一个在 A 时间点执行性能很高的执行计划，在 B 时间点由于数据内容发生了变化，它的性能可能就很低，查询优化器则随时都能根据数据的变化调整执行计划，而数据库用户则只能手动更改执行方案，和查询优化器相比，它的时效性比较低。
- 查询优化器和数据库用户之间的计算能力不同，目前计算机的计算能力已经大幅提高，在执行数值计算方面和人脑相比具有巨大的优势，查询优化器对一个语句进行优化时，可以从几百种执行方案中选出一个最优的方案，而人脑要全面地计算这几百种方案，需要的时间远远要长于计算机。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

因此，查询优化器是提升查询效率非常重要的一个手段，虽然一些数据库也提供了人工干预生成查询计划的方法，但是通常而言查询优化器的优化过程对数据库开发人员是透明的，它自动地进行逻辑上的等价变换、自动地进行物理路径的筛选，极大地解放了数据库应用开发人员的“生产力”。

通常数据库的查询优化方法分为两个层次：

- 基于规则的查询优化（逻辑优化，Rule Based Optimization，简称 RBO）。
- 基于代价的查询优化（物理优化，Cost Based Optimization，简称 CBO）。

逻辑优化是建立在关系代数基础上的优化，关系代数中有一些等价的逻辑变换规则，通过对关系代数表达式进行逻辑上的等价变换，可能会获得执行性能比较好的等式，这样就能提高查询的性能；而物理优化则是在建立物理执行路径的过程中进行优化，关系代数中虽然指定了两个关系如何进行连接操作，但是这时的连接操作符属于逻辑运算符，它没有指定以何种方式实现这种逻辑连接操作，而查询执行器是不“认识”关系代数中的逻辑连接操作的，我们需要生成多个物理连接路径来实现关系代数中的逻辑连接操作，并且根据查询执行器的执行步骤，建立代价计算模型，通过计算所有的物理连接路径的代价，从中选择出“最优”的路径。

1.2　逻辑优化

逻辑优化是建立在关系代数基础之上的优化形式，下面通过介绍关系模型的理论知识来认识逻辑优化。

1.2.1　关系模型

关系数据库采用关系模型来描述数据，每个数据库是一个“关系”的集合，这个“关系”就是我们通常所谓的表，其形态类似于一个二维数组，我们称其中的一行为一个“N-元组”，通常简称为“元组”，其中的一列代表的是一个“属性”，所有属性的值最终组成了“域”。

例如有如下的关系：

```
STUDENT(sno, sname, ssex);
COURSE(cno, cname, tno);
SCORE(sno, cno, degree);
TEACHER(tno, tname, tsex);
```

每个关系的实例如图 1-2 所示。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

STUDENT		
sno	sname	ssex
5	zs	男
3	ls	女
2	ww	男
4	zl	男
1	lq	女

COURSE		
cno	cname	tno
1	English	2
2	Math	5
3	Data	3
4	Design	5
5	Phys	6

SCORE		
sno	cno	degree
2	1	60
3	2	50
1	3	80
1	5	90
4	4	85
3	3	99
5	1	78

TEACHER		
tno	tname	tsex
1	Jim	男
2	Tom	女
3	Lucy	男
4	Dadge	女
5	Benny	男

图 1-2　关系实体的数据

其中 STUDENT 代表了一个关系，而 sno、sname、ssex 则代表的是这个关系的属性，STUDENT 关系中共包含 5 个元组。

在关系模型中，为了对关系、元组、属性等进行操作，定义了两种形式化的语言，分别是关系代数和关系演算。关系代数从逻辑的角度定义了对数据进行操作的方法，而关系演算则是描述性的，它准确地刻画需要获得的结果而不关心获得结果的过程。

关系代数的操作主要包含 5 个基本操作符，分别是选择（σ）、投影（Π）、笛卡儿积（×）、并集（∪）、差集（－），其中并集操作、差集操作、笛卡儿积操作来自集合论，选择和投影操作则是关系代数所特有的操作，这些基本操作是关系代数的基石，缺一不可，通过这些基本操作还可以衍生出一些其他比较重要的操作（可以用上述的 5 个基本操作将其表达出来），其中最重要的是交集操作（∩）和连接操作（⋈）。如图 1-3 所示分别是投影、选择、和笛卡儿积操作的示例。

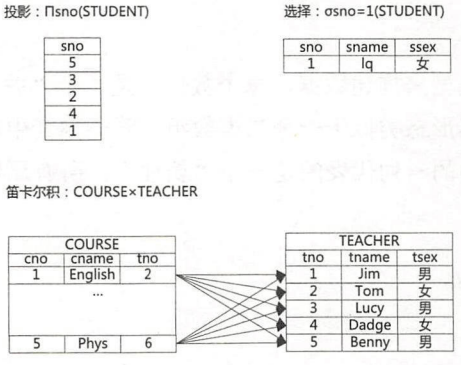


图 1-3　投影、选择、笛卡儿积的示例

另外，结合数据库的实际使用情况，通常数据库会对关系代数的逻辑运算符做一些扩展，

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

例如外连接操作（左外连接⟕、右外连接⟖、全外连接⟗、半连接⋉、反半连接▷）、聚集和分组操作（γ）等，这些都不符合经典的关系代数理论，但是它们在数据库应用开发的过程中却经常被用到，扩展操作的示例分别如图 1-4 和图 1-5 所示。



图 1-4　扩展连接操作的示例

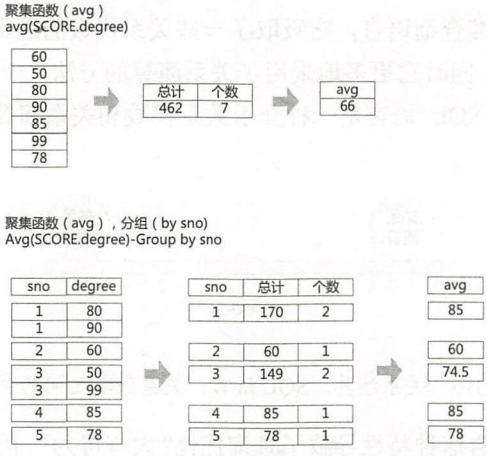


图 1-5　聚集和分组操作的示例

连接操作中左侧的表通常可以称为外表、左表或 LHS 表，右侧的表通常称为内表、右表或 RHS 表，而在外连接中还可以根据是否补 NULL 值进行区分，比如左连接，它的外表无须补 NULL 值，因此是 Nonnullable-side 的表，而左连接的内表需要补 NULL 值，因此是 Nullable-side 的表。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

关系演算和关系代数不同，它从更高的层次来描述计算结果，完全不关心计算的过程，其中基于元组的关系演算可以称为元组关系演算，基于属性的关系演算称为域关系演算，元组关系演算包含的操作符如下：

- 存在量词（∃）和全称量词（∀）。
- 比较操作符（>，>=，<，<=，=，!=）。
- 逻辑操作符（¬，∧，∨，=>）。

关系演算的表达能力和关系代数是等价的，我们用元组关系演算的方式来实现选择、投影等几个关系代数表达式，如表 1-1 所示。

表 1-1　关系代数和关系演算的等价对照表

关系代数	元组关系演算
Πsno(STUDENT)	{P|∃(t) ∧ (STUDENT(t) ∧P.sno = t.sno)}
σ sno=1(STUDENT)	{P|∃(t) ∧ (STUDENT(t) ∧ P.sno = t.sno ∧ P.sname=t.sname∧P.ssex = t.ssex ∧t.sno = 1)}
Πcno,cname,course.tno,teacher.tno,tname,tsex(COURSE×TEACHER)	{P|∃(tc) ∧ ∃(tt) ∧ COURSE(tc) ∧TEACHER(tt) ∧ P.cno = tc.cno ∧ P.cname=tc.cname ∧ P.tno1=tc.tno ∧ P.tno2=tt.tno ∧ P.tname=tt.tname ∧ P.tsex=tt.tsex}

SQL 作为数据库的标准查询语言，它吸取了一些关系代数的逻辑操作符，但是放弃了关系代数中“过程化”的特点，同时它更多地采用了关系演算的方法，一个 SQL 语句通常是对执行结果的描述，因此我们说 SQL 语言是一种介于关系代数和关系演算之间的描述性语言，如图 1-6 所示。

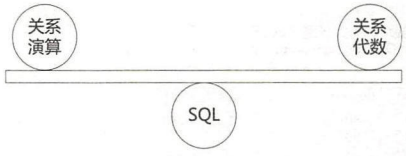


图 1-6　关系演算、SQL 语言、关系代数之间的关系

SQL 语言是描述性语言这种特性导致了查询优化“大有可为”，因为它只规定了“WHAT”，而没有规定“HOW”，不同的获取结果的方法代价相差可能极大，因此数据库的查询优化就变得极为重要了。

既然逻辑优化是建立在关系代数等价变换基础上的优化，下面我们先来总结一下关系代数有哪些等价变换的规则。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

规则 1：交换律：

$A \times B \quad == \quad B \times A$

$A \bowtie B \quad == \quad B \bowtie A$

$A \bowtie_F B \quad == \quad B \bowtie_F A$ ..... 其中 F 是约束条件

$\Pi_p(\sigma_F(B)) == \sigma_F(\Pi_p(B))$ ..... 其中 $F \in p$

规则 2：结合律：

$(A \times B) \times C \quad == \quad A \times (B \times C)$

$(A \bowtie B) \bowtie C \quad == \quad A \bowtie (B \bowtie C)$

$(A \bowtie_{F_1} B) \bowtie_{F_2} C \quad == \quad A \bowtie_{F_1} (B \bowtie_{F_2} C)$ ..... 其中 F_1 和 F_2 是约束条件

规则 3：分配律

$\sigma_F(A \times B) == \sigma_F(A) \times B$ ..... 其中 $F \in A$

$\sigma_F(A \times B) == \sigma_{F_1}(A) \times \sigma_{F_2}(B)$ ..... 其中 $F = F_1 \cup F_2$，$F_1 \in A, F_2 \in B$

$\sigma_F(A \times B) == \sigma_{F_x}(\sigma_{F_1}(A) \times \sigma_{F_2}(B))$ ..... 其中 $F = F_1 \cup F_2 \cup F_x$，$F_1 \in A, F_2 \in B$

$\Pi_{p,q}(A \times B) == \Pi_p(A) \times \Pi_q(B)$ ..... 其中 $p \in A$，$q \in B$

$\sigma_F(A \times B) == \sigma_{F_1}(A) \times \sigma_{F_2}(B)$ ..... 其中 $F = F_1 \cup F_2$，$F_1 \in A, F_2 \in B$

$\sigma_F(A \times B) == \sigma_{F_x}(\sigma_{F_1}(A) \times \sigma_{F_2}(B))$ ..... 其中 $F = F_1 \cup F_2 \cup F_x$，$F_1 \in A, F_2 \in B$

规则 4：串接律

$\Pi P=p_1,p_2,\ldots p_n(\Pi\, Q=q_1,q_2,\ldots q_n(A)) == \Pi\, P=p_1,p_2,\ldots p_n(A)$ ..... 其中 $P \subseteq Q$

$\sigma_{F_1}(\sigma_{F_2}(A)) == \sigma_{F_1 \wedge F_2}(A)$

上面的规则并不能把所有的情况都列举出来，如果读者有集合论和数理逻辑的基础，那么就能灵活地理解和运用这些规则，例如，如果对 $\sigma_{F_1}(\sigma_{F_2}(A)) == \sigma_{F_1 \wedge F_2}(A)$继续推导，那么就可以获得：

$$\sigma_{F_1}(\sigma_{F_2}(A)) == \sigma_{F_1 \wedge F_2}(A) == \sigma_{F_2 \wedge F_1}(A) == \sigma_{F_2}(\sigma_{F_1}(A))$$

也就是说选择操作满足 $\sigma_{F_1}(\sigma_{F_2}(A)) == \sigma_{F_2}(\sigma_{F_1}(A))$这样的交换律，集合论和数理逻辑都属于离散数学中的内容，因此感兴趣的读者可以参阅一些离散数学的相关资料。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

1.2.2 逻辑优化示例

下面来看一个通过关系代数等价变换规则进行优化的示例，如果要获得编号为 5 的老师承担的所有的课程名字，我们可以给出它的关系代数表达式：

Πcname (σTEACHER.tno=5∧TEACHER.tno=COURSE.tno (TEACHER×COURSE))

由于笛卡儿积是一个比较“重”的操作，如果将选择操作优先做，先把关系上的数据筛选掉一部分，这样就能够降低笛卡儿积的计算量，因此应用规则 $\sigma_F(A \times B) == \sigma_{F_1}(A) \times \sigma_{F_2}(B)$将选择下推，关系代数表达式变换成：

Πcname (σTEACHER.tno=COURSE.tno (σTEACHER.tno=5(TEACHER)×COURSE))

从上面的表达式可以看出通过将约束条件 TEACHER.tno=5 下推，优先对 TEACHER 表进行过滤，降低了笛卡儿积操作符 LHS 的关系的大小，降低了笛卡儿积操作的计算量，我们还能根据投影的串接律将投影下推，从垂直方向上缩小关系的大小：

Πcname (σTEACHER.tno=COURSE.tno (σTEACHER.tno=5(TEACHER)× Πcname, tno(COURSE)))

在投影下推的时候，由于约束条件 TEACHER.tno=COURSE.tno 中还需要使用到 COURSE.tno，因此在应用串接律的时候需要对 COURSE.tno 也进行投影，这样笛卡儿积的 RHS 中间结果也缩小了。

如果再仔细分析上面的关系代数表达式，还可以发现在约束条件 TEACHER.tno = COURSE.tno∧TEACHER.tno = 5 中有一个隐含的等价推理，我们可以从中推导出一个新的约束条件 COURSE.tno = 5，因为在产生笛卡儿积的过程中，COURSE 关系中如果有属性 tno 不等于 5 的元组，它最终也不会被输出到连接结果中，因此还可以进一步优化这个关系代数表达式，应用新的约束条件 COURSE.tno = 5：

Πcname (σTEACHER.tno=COURSE.tno
(σTEACHER.tno=5(TEACHER)× Πcname, tno(σCOURSE.tno=5(COURSE))))

这样笛卡儿积操作 LHS 和 RHS 的关系从水平方向和垂直方向都缩小了。这个关系表达式还可以进一步优化，因为现在经过 $\sigma_{TEACHER.tno=5}(TEACHER)$选择产生的结果关系中 tno 的值一定是 5，而经过 $\sigma_{COURSE.tno=5}(COURSE)$选择产生的结果关系中 tno 的值也一定是 5，也就是说经过选择操作下推之后，TEACHER.tno = COURSE.tno 已经隐含是一个永远为 TRUE 的约束条件，因此关系代数表达式可以变换成：

Πcname (σTEACHER.tno=5(TEACHER)× Πcname, tno(σCOURSE.tno=5(COURSE)))

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

从图 1-7 和图 1-8 中可以看出，应用关系代数的等价转换之后，关系代数表达式的计算量降低了。

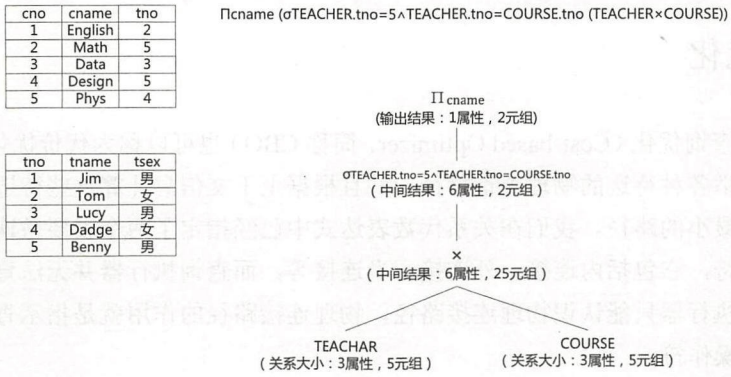


图 1-7　关系代数示例，优化前的关系代数表达式

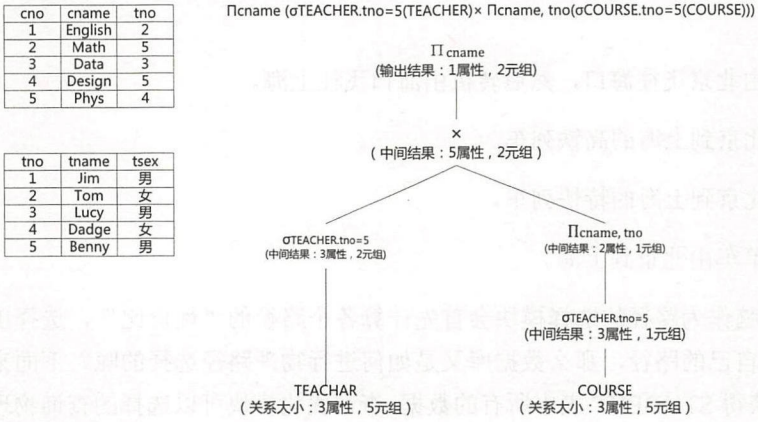


图 1-8　关系代数示例，优化后的关系代数表达式

基于上述示例中的关系代数的等价变换，我们可以获得两个启发式的规则，它们分别从水平方向和垂直方向上尽早地缩小笛卡儿积的中间结果：

- 尽量将选择操作下推到下层节点来做。
- 尽量在叶子节点上使用投影缩小中间结果。

上面的示例看似完美是因为它总能将过滤条件一推到底，然而在数据库的实现过程中却要面临很多障碍，比如数据库除了基本的关系代数操作之外还扩展出了外连接、聚集操作，在内

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

连接中能够进行下推的约束条件，如果换成外连接就不一定能够下推，基于内连接能够做的等价推理换成外连接也不一定等价，这极大地增加了数据库查询优化的难度。

1.3 物理优化

基于代价的查询优化（Cost-based Optimizer，简称 CBO）也可以称为代价优化、物理优化，其主要流程是枚举各种待选的物理查询路径，并且根据上下文信息计算这些待选路径的代价，进而选择出代价最小的路径。我们在关系代数表达式中已经指定了两个表要做连接，这种连接操作是逻辑操作符，它包括内连接、外连接、半连接等，而查询执行器并无法直接执行这些逻辑操作符，查询执行器只能认识物理连接路径，物理连接路径的作用就是指示查询执行器以何种方式实现逻辑操作符。

生活中不乏类似物理优化的例子，正所谓条条大路通罗马。例如公司员工需要由北京去上海出差，去上海出差就好比一个逻辑操作符，但是以何种方式去上海出差并没有设定，假如待选的路径如下。

A：乘飞机由北京飞往海口，然后转机由海口飞往上海。

B：乘坐由北京到上海的高铁列车。

C：乘坐由北京到上海的特快列车。

D：骑共享单车由北京到上海。

该员工的大脑作为路径的选择模块会首先计算各个路径的“性价比”，选择出“性价比”最高的、最适合自己的路径，那么数据库又是如何进行物理路径选择的呢？下面来看一个物理路径的例子，要获得 STUDENT 表中所有的数据，查询优化模块可以选择的查询物理路径如下。

A：扫描表的全部数据页面来获得所有的元组，并在所有元组上进行选择和投影。

B：如果表上有索引，并且约束条件满足索引的要求，可以尝试扫描索引，并在索引扫描产生的结果上做选择和投影。

C：如果有特殊的约束条件，还可以尝试位图扫描或 TID 扫描等，并在扫描的结果上做选择和投影。

数据库无法从感性的角度来衡量哪条物理路径的代价低，因此它需要构建一个量化的模型，这个代价模型需要从两个方面来衡量路径的代价：

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

执行代价 = IO 代价 + CPU 代价

产生 IO 代价的原因是因为数据是保存在磁盘上的，要对数据进行处理，需要将数据从磁盘加载到主存，另外在数据需要排序、建立 hash 表、物化的时候还可能需要将处理后的数据写入磁盘，这些都是 IO 代价，数据库要计算一个查询的 IO 代价存在一些困难：

- 磁盘 IO 到底是什么样的代价基准，由于磁盘种类的不同，它的读写效率不同，如果有些数据挂载在机械磁盘上，而有些数据挂载在 SSD 磁盘上，那么不同磁盘上的数据的 IO 效率相差非常大，数据库如何区分这种区别呢？
- 数据库本身是有缓存系统的，假如某个表上有一些数据已经保存在缓存中了，这些数据在对表进行扫描的时候就不会产生 IO，因此要想计算准确的 IO 代价，数据库还需要知道一个表中有多少页面在缓存中，有多少页面没有在缓存中，但是缓存中的页面可能随时地换入换出，数据库是否有能力实时地记录这种变化呢？
- 磁盘本身也有磁盘的缓存系统，在磁盘上随机读写和顺序读写的效率也不同，那么顺序读写和随机读写的效率差别如何量化呢？机械磁盘上顺序读写和随机读写的性能差别可能差距比较大，而 SSD 磁盘上的顺序读写和随机读写的性能差距则相对较小，数据库如何量化这种区别呢？

产生 CPU 代价的原因是选择、投影、连接都需要进行大量的运算，尤其是像聚集函数这样 CPU 密集型的操作符，而 CPU 代价的计算和 IO 代价一样也面临一些问题：

- 和 IO 代价类似，CPU 也拥有很多型号，各种型号间的性能不同，是否需要量化一个 CPU 代价的基准单位？
- CPU 为了加快执行速度，它有多级的 cache，这些 cache 比数据库的缓存（内存）效率更高，在计算 CPU 代价的时候是否需要考虑这些 cache 带来的性能优化呢？
- 不同的表达式在执行的时候产生的代价不同，有些表达式的代价大，有些表达式的代价小，数据库如何量化这些表达式产生的代价呢？

要想计算物理路径的代价，数据库还需要对数据的分布情况有一个了解，因为无论是 IO 代价还是 CPU 代价，都是建立在对数据处理的基础之上的，数据的分布情况也会从很大程度上对代价产生影响：

- 相同的数据在不同的分布下所带来的开销不同，例如数据在有序的情况下和无序的情况下，如果要执行一个排序的操作，那么就可能一个需要排序，而另一个不需要排序，开销肯定是不同的，再例如同样一份数据，它在磁盘上的存储是稀疏的还是紧凑的对 IO 代价的影响也非常大。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

- 相同的数据在面临不同的选择操作时，它的开销也不同，比如选择操作要处理数据中的高频值，相对而言它需要的计算就多一些，因此代价也会高一些。

总之，数据库很难给出一个“准确”的代价模型来描述所有的情况，计算代价的目的是在物理路径之间进行挑选，它只需要能够用于比较物理路径的优劣就够了，虽然大部分数据库都采用了 IO 代价和 CPU 代价来衡量物理路径的代价，但具体的实现细节则千差万别，一个数据库的代价模型需要不断地“修炼”才能接近完美。

1.3.1 物理优化的 4 个“法宝”

关系的本身可以视为一个集合或者包，这种数据结构对数据的分布没有设定，为了提升计算的性能，我们需要借助一些数据结构或算法来对数据的分布做一些预处理，在物理优化的过程中有 4 个非常重要的数据结构或算法贯穿其中，下面简单介绍一下这些方法。

1.3.1.1 B+树

如果要查询一个表中的数据，最简单的办法自然是将表中的数据全部遍历一遍，但是随着当前数据量越来越大，遍历表中数据的代价也越来越高，而 B+树就成了我们高效地遍历数据的有力武器。

1970 年，R.Bayer 和 E.mccreight 提出了一种适用于外查找的树，它是一种平衡的多叉树，称为 B 树，B 树就是在表的数据上建立一个“目录”，类似于书籍的目录，这样就能快速地定位到要查询的数据。

B 树具有如下性质：

- 除了根节点之外，每个节点至少拥有 m/2 个子节点，也就是说每个节点上最少半满，至多全满，多于全满会发生节点的分裂，少于半满会发生节点的合并，这种半满和全满的属性保证了在增加或者删除叶子节点的过程中，不会频繁地合并和分裂。
- 所有的叶节点都在同一层上，也就是说查找所有叶子节点的复杂度是相同的，都等于树高，由于 B 树是一棵多叉树，通常树的高度不会超过 4，因此查找一个叶子节点的复杂度不高。
- 有 k 棵子树的分支节点则存在 k-1 个关键码，关键码按照递增次序进行排列，也就是说树中的在同一层上的节点是有序的，也就是说叶子节点也是有序的。

PostgreSQL 的数据库使用的是基于 Lehman 和 Yao 的论文进行改进的 B+树，它在原来 B 树的基础上增加了“下一个节点的指针”和“页内最大值”，这样能提高 B 树的使用效率。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

B+树作为一种数据结构和查询优化器本身没有直接的关系，但是数据库通常会建立基于B+树的索引，而在查询优化的过程中，索引扫描、位图扫描都会涉及这种 B+树类型的索引，在索引扫描的过程中会根据 B+树的情况计算代价，不同的 B+树索引扫描的代价是不同的。例如，STUDENT 表上有一个基于 B+树的主键索引，因此通过 sno=1 这样的约束条件查询 B+树，它的代价等于 B+树的树高，而顺序扫描则需要遍历整个 STUDENT 表，因此 B 树对单值的查询、基于范围的约束条件通常都有比较好的查询效率。

1.3.1.2　Hash 表

Hash 表也是一种对数据进行预处理的方法，PostgreSQL 数据在多个地方使用了 Hash 表或借用了 Hash 表的思想，它在查询优化器中有如下使用方式：

1）借用 Hash 表可以实现分组操作，因为 Hash 表天然就有对数据分类的功能。

2）借用 Hash 可以建立 Hash 索引，这种索引适用于等值的约束条件。

3）物理连接路径中 Hash Join 是非常重要的一条路径，它对内表建立 Hash 表，外表的元组在 Hash 表中进行探测。

1.3.1.3　排序

排序也是一种对数据进行预处理的方法，它主要用在如下几个方面。

1）借用排序可以实现分组操作，因为经过排序之后，相同的数据都聚集在一起，因此它可以用来实现分组。

2）B 树索引的建立需要借助排序来实现，PostgreSQL 数据库采用堆的方式对数据进行存储，而 B 树索引的叶子节点是有序的，因此需要先将数据进行排序，而后在有序的数据上建立 B 树索引。

3）物理连接路径 MergeJoin 路径需要借助排序实现，MergeJoin 需要先对连接操作中的内表和外表进行排序，然后才能进行“归并”。

4）数据库中的 Order By 操作需要借助排序实现，Order By 本身的语义就是对数据进行排序。

在数据量比较小时，数据可以全部加载到内存，这时使用内排序就能完成排序的工作，而当数据量比较大时，则需要使用外排序才能完成排序的工作，因此在计算排序的代价时需要根据数据量的大小以及可使用的内存的大小来决定排序的代价。

1.3.1.4　物化

物化就是将扫描操作或者连接操作的结果保存起来，这种保存是有代价的，因为如果扫描

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

操作或者连接操作产生的中间结果比较大，就可能需要将中间结果写入外存，这会产生 IO 代价。

物化的优点是如果数据可以一次产生多次使用，那么就可以将这个中间结果保存下来多次利用，例如对 STUDENT 表和 SCORE 表做连接，如果 SCORE 表经过扫描之后，只有 5%的数据作为中间结果，其他 95%的数据都被过滤掉了，那么就可以考虑将这 5%的数据物化起来，这样 STUDENT 表的每条元组只和这 5%的数据进行连接就可以了。

中间结果是否物化主要取决于代价计算的模型，通常物理优化生成物理路径时会比较物化和不物化两条路径的代价，最终选择代价较低的一个。

1.3.2 物理路径的生成过程

数据库中的物理路径大体上可以分为扫描路径和连接路径，扫描路径是针对单个关系的，它可以用来实现关系代数中的选择和投影，而连接路径对应的是两个表做连接操作，因此它可以用来实现关系代数中的笛卡儿积。

1.3.2.1 物理路径的分类

如果要获得一个关系中的数据，最基础的方法就是将关系中的所有数据都遍历一遍，从中挑选出符合条件的数据，如图 1-9 所示，这种方式就是顺序扫描路径，顺序扫描路径的优点是具有广泛的适用性，各种“关系”都可以用这种方法，它的缺点自然是代价通常比较高，因为要把所有的数据都遍历一遍。



图 1-9 顺序扫描示意图

如果将数据做一些预处理，比如建立一个索引，如果要想获得一个表的数据，可以通过扫描索引获得所需数据的“地址”，然后通过地址将需要的数据获取出来，尤其是在选择操作带有约束条件的情况下，在索引和约束条件共同的作用下，关系中的有些数据就不用再遍历了，因为通过索引就很容易知道这些数据是不符合约束条件的，更有甚者，因为索引上也保存了数据，它的数据和关系中的数据是一致的，因此如果索引上的数据能满足要求，只需扫描索引就可以获得所需数据了，也就是说在扫描路径中还可以有索引扫描路径和快速索引扫描路径两种方式，如图 1-10 所示。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

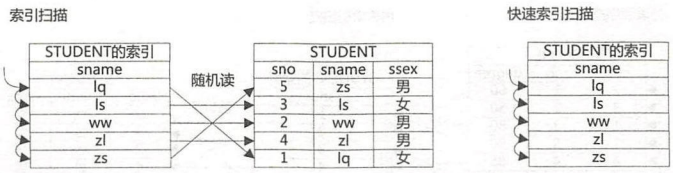


图 1-10　索引扫描和快速索引扫描示意图

索引扫描路径带来一个问题，它可能会带来大量的随机读，因为索引中记录的是数据元组的地址，索引扫描通过扫描索引获得元组地址，然后通过元组地址访问数据，索引中保存的“有序”的地址，到数据中就可能是随机的了。为了解决这个问题，又增加了位图扫描，它通过位图将地址保存起来，把地址收集起来之后，让地址变得有序，这样就通过中间的位图把随机读消解掉了，如图 1-11 所示。

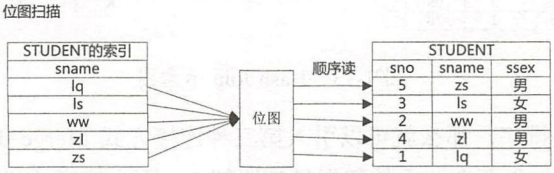


图 1-11　位图扫描示意图

当然，扫描的过程中还会结合一些特殊的情况使用一些非常高效的扫描路径，比如 TID 扫描路径，TID 实际上是元组在磁盘上的存储地址，我们能够根据 TID 直接获得元组，这样查询的效率就非常高了。

扫描路径通常是执行计划中的叶子节点，也就是在最底层对表进行扫描的节点（也可能扫描节点的下层又是一个子执行计划），扫描路径就是为连接路径做准备的，扫描出来的数据就可以给连接路径来实现连接操作了。

要对两个关系做连接，受笛卡儿积的启发，可以用一个算法复杂度是 $O(m * n)$的方法来实现，我们叫它 Nestlooped Join 方法，这种方法虽然复杂度比较高，但是和顺序扫描一样，胜在具有普适性。如果 Nestlooped Join 的内表的路径是一个索引扫描路径，那么算法的复杂度就会降下来，索引扫描的算法复杂度是 $O(\log n)$，因此如果 Nestlooped Join 的内表是一个索引扫描，它的整体的算法复杂度就变成了 $O(m\log n)$，看上去这样也是可以接受的，如图 1-12 所示。

如果 Nestlooped Join 的内表上没有索引，那么我们是否可以将内表的数据做一些处理，让算法的复杂度降低下来呢？答案是肯定的，在内表数据量不多的情况下，可以建立一个 Hash 表，如图 1-13 所示，这样由外表驱动在内表的 Hash 表上探测约束条件，假设 Hash 表有 N 个桶，内表数据均匀地分布在各个桶中，那么 Hash Join 的时间复杂度就是 $O(m * n / N)$。

非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

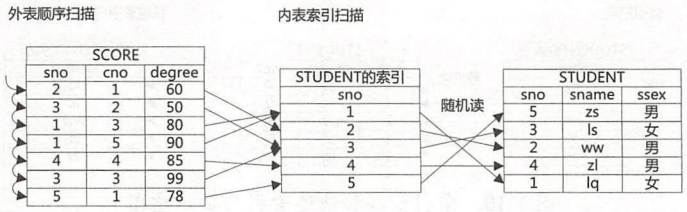


图 1-12　Nested Loop Join 示意图

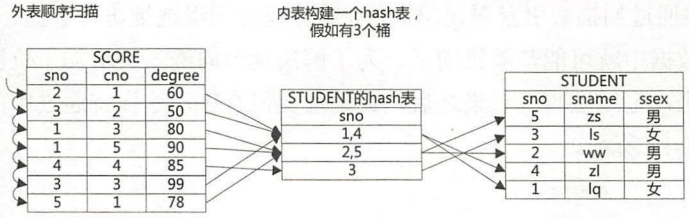


图 1-13　Hash Join 示意图

如果将两个关系先排序，那么就可以引入第三种连接方式 Merge Join，这种连接方式的代价主要浪费在排序上，如果两个关系的数据量都比较小，那么排序的代价是可控的，Merge Join 就是适用的，另外如果关系上有有序的索引，那么就可以不用单独排序了，这样也比较适用于 Merge Join，如图 1-14 所示。

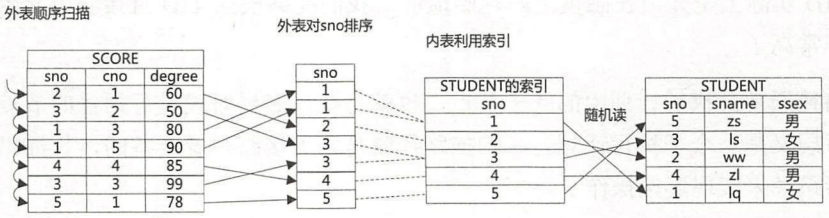


图 1-14　Merge Join 示意图

综上所述，我们对物理路径有了基本的了解，物理扫描路径主要有顺序扫描路径、（快速）索引扫描路径、位图扫描路径、TID 扫描路径等，而物理连接路径主要有 Nestlooped Join、Hash Join 和 Merge Join，这些路径的生成都会或多或少地使用到物理优化的 4 个“法宝”，因此读者可以将它们结合在一起来理解物理优化路径生成的过程。

1.3.2.2　路径搜索的方法

物理路径的种类越多，挑选最优路径的难度就越大，例如有 3 个表要做连接操作，每个表上有 3 个扫描路径，那么扫描路径的组合就有 27 种可能，由于表之间的连接顺序可以交换，3

非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

个表可能产生的连接顺序有 12 种情况，每个连接路径可能的物理连接路径是 9 种情况，那么最终要生成 3 个表的连接路径“树”，共需要计算 27×12×9 = 2916 种情况。然后从中选出最优的那个路径“树”，如果再增加新的表参与连接操作，那么物理执行计划的解空间就会以几何级数的方式不断地增长，因此物理路径的搜索方法就非常重要了，通常物理路径的搜索有以下几种方法：

1）物理路径的搜索方法中最常用的是自底向上的一种方法，代表性的就是 System-R 系统所使用的模型——动态规划方法，这种方法把查找最优执行计划问题划分为子问题，用最优的子问题不断地向上迭代，最终获得最优解。

2）还有自顶向下的方法，这种方法对逻辑优化和物理优化没有明显的界限，它先通过自顶向下的方式构造逻辑查询树以及物理查询树，和自底向上的方法不同，它不是通过子问题的最优解迭代出整个问题的最优解，而是通过先构建出逻辑的整个查询树，然后再迅速地枚举各种物理路径。

3）随机搜索的方法也是一种重要的物理路径搜索方法，由于自底向上的方法和自顶向下的方法都需要遍历所有的解空间，因此在参与连接的表比较多的情况下，可以尝试采用随机的搜索算法，例如遗传算法，这种算法的优点是在有限的解空间内尝试取得最优的连接路径，它的效率是可控的，而缺点则是最优解是有限解空间的最优解，在整体空间上它可能只是一个局部的最优解。

PostgreSQL 数据库采用了其中的两种方法，一种是在表的数量比较少的情况下采用基于 System-R 系统的动态规划方法，另一种是在表的数量比较多的情况下启用遗传算法，在第 7 章中我们会对这两种方法进行介绍。对自顶向下的方法感兴趣的读者可以参考 Pivotal 公司开源的查询优化器 ORCA 的实现方法。

1.4　文件介绍

PostgreSQL 数据库的查询优化的代码在 src/backend/optimizer 目录下，其中有 plan、prep、path、geqo、util 共 5 个子目录，plan 是总入口目录，它调用了 prep 目录进行逻辑优化，调用 path、geqo 目录进行物理优化，util 目录是一些公共函数，供所有目录使用。

如图 1-15 所示，Plan 模块为总调用模块，Prep 和 Path 被它调用。在执行中，从 Plan 模块入口，先调用 Prep 模块进行预处理，再调用 Path 模块进行优化。Path 模块中有开关，指示是否启用遗传算法进行优化，如果启用，且连接的表超过 11，就调用 geqo 目录中的遗传算法进

非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

行优化。util 模块为辅助工具模块，提供其他模块使用的工具函数。

prep 目录主要处理逻辑优化中的逻辑重写的部分，对投影、选择条件、集合操作、连接操作都进行了重写，如图 1-16 所示。

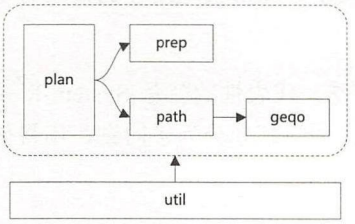


图 1-15　PostgreSQL 数据库查询优化文件目录结构图

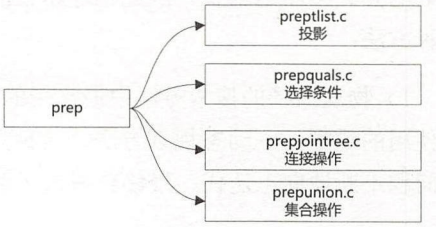


图 1-16　PostgreSQL 数据库库查询优化 prep 目录结构图

path 目录则主要是生成物理路径的部分，包括生成扫描路径、连接路径等，如图 1-17 所示。

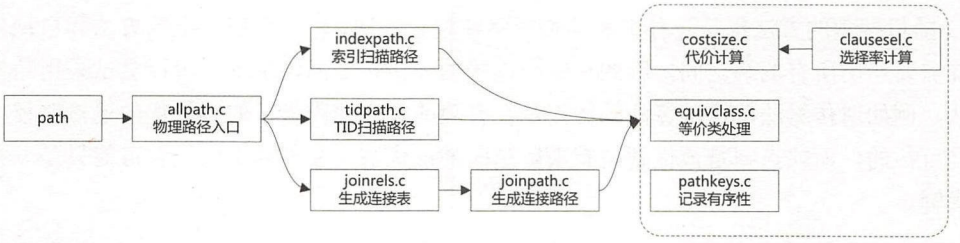


图 1-17　PostgreSQL 数据库查询优化 path 目录结构图

geqo 目录主要是实现了一种物理路径的搜索算法——遗传算法，通过这种算法可以处理参与连接的表比较多的情况，在第 7 章中会详细地进行介绍，utils 目录则提供了大量的公共函数，其他各个目录中均可能会调用这些函数。

1.5　示例的约定

本书示例在逻辑优化阶段主要使用了 STUDENT、COURSE、SCORE、TEACHER 这几个表作为示例，下面给出它们的定义：

```
-- STUDENT（学号，学生姓名，学生性别）
CREATE TABLE STUDENT(sno INT primary key, sname VARCHAR(10), ssex INT);
-- COURSE（课程编号，课程名，教师编号）
```


非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
CREATE TABLE COURSE(cno INT primary key, cname VARCHAR(10), tno INT);
-- SCORE（学号，课程编号，分数）
CREATE TABLE SCORE(sno INT, cno INT, degree INT);
-- TEACHER（教师编号，教师姓名，教师性别）
CREATE TABLE TEACHER(tno INT primary key, tname VARCHAR(10), tsex INT);
```

在物理优化阶段则主要使用 TEST_A、TEST_B、TEST_C……作为示例，每个表的列名约定为 a、b、c、d……，下面也给出它们的定义：

```
CREATE TEST_A(a INT, b INT, c INT, d INT,……);
CREATE TEST_B(a INT, b INT, c INT, d INT,……);
CREATE TEST_C(a INT, b INT, c INT, d INT,……);
CREATE TEST_D(a INT, b INT, c INT, d INT,……);
```

1.6　小结

关系数据库的查询优化通常分为逻辑优化和物理优化。逻辑优化是基于关系代数的等价的逻辑变换，关系代数中有大量的逻辑等价规则，可以利用这些规则尝试将选择操作和投影操作尽量下推，缩小查询中的中间结果以提高执行效率；物理优化则是通过代价估算的方式挑选代价比较低的物理路径，根据物理路径的性质又可以分成扫描路径和连接路径，在生成物理路径的过程中通常可以选择动态规划方法等最优路径算法来获得对物理路径进行搜索。

虽然在本章中已经介绍了部分查询优化涉及的基础理论，但是这还远远不够，有兴趣的读者可以参阅《数据库系统实现》《数据库系统概念》等理论专著的相关章节来了解查询优化的基础理论，另外也可以查阅相关的论文来了解学术界对查询优化的不断改进。

非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

2 第2章 查询树

一个 SQL 查询语句在经过了词法分析、语法分析和语义分析之后会形成一棵查询树，这棵查询树实际上就对应了一个关系代数表达式，它是查询优化器的输入，贯穿了查询优化的整个过程。所谓“工欲善其事，必先利其器”，在开始对查询优化器的代码进行分析之前，对查询树必须要有一定的了解，下面就开始分析查询树的结构，同时也介绍一下查询树涉及的其他的数据结构。

2.1 Node 的结构

PostgreSQL 数据库中的结构体采用了统一的形式，它们都是基于 Node 结构体进行的“扩展”，Node 结构体中只包含一个 NodeTag 成员变量，NodeTag 是 enum（枚举）类型。

```
typedef struct Node
{
    NodeTag type;
} Node;
```


非卖品，仅供非商业用途或交流学习使用 严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

其他的结构体则利用 C 语言的特性对 Node 结构体进行扩展，所有结构体的第一个成员变量也是 NodeTag 枚举类型，例如在 List 结构体里，第一个成员变量是 NodeTag，它可能的值是 T_List、T_intList 或者 T_OidList，这样就能分别指代不同类型的 List。

```
typedef struct List
{
    NodeTag  type;    /* T_List, T_IntList, or T_OidList */
    int  length;
    ListCell  *head;
    ListCell  *tail;
} List;
```

而 Query 结构体也是以 NodeTag 枚举类型作为第一个变量，它的取值为 T_Query。

```
typedef struct Query
{
    NodeTag       type;
    CmdType       commandType;  /* select|insert|update|delete|utility */
    QuerySource querySource;    /* where did I come from? */
    ......
} Query;
```

这样无论是 List 结构体的指针，还是 Query 结构体的指针，我们都能通过 Node 结构体的指针（Node*）来表示，而在使用对应的结构体时，则通过查看 Node 类型的指针中的 NodeTag 枚举类型就可以区分出该 Node 指针所代表的结构体的实际类型。

2.2　Var 结构体

Var 结构体表示查询中涉及的表的列属性，在 SQL 语句中，投影的列属性、约束条件中的列属性都是通过 Var 来表示的，在语法分析阶段会将列属性用 ColumnRef 结构体来表示，在语义分析阶段会将语法树中的 ColumnRef 替换成 Var 用来表示一个列属性。下面来看一下 Var 结构体中各个变量的具体含义。

varno：用来确定列属性所在的表的“编号”，这个编号源自 Query（查询树）中的 rtable 成员变量，查询语句中涉及的每个表都会记录在 rtable 中，而其在 rtable 中的“编号”（也就是处于链表的第几个，从 1 开始计数，这个编号用 rtindex 表示）是唯一确定的值，在逻辑优化、物理优化的过程中可能 varno 都是 rindex，而在生成执行计划的阶段，它可能不再代表 rtable 中的编号，这在第 10 章中会进行介绍。

非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

varattno/vartype/vartypmod：varattno 确定了这个列属性是表中的第几列，vartype 和 vartypmod 则和列属性的类型有关。在创建表的时候，PostgreSQL 数据库会按照 SQL 语句中指定的列的顺序给列属性编号，并将编号记录在 PG_ATTRIBUTES 系统表中，同时会将 SQL 语句指定的列的类型也记录到 PG_ATTRIBUTES 中，因此可以说 varattno、vartype、vartypemod 都是取自 PG_ATTRIBUTES 系统表中的。

varlevelsup：确定了列属性对应的表所在的层次，这个层次值是一个相对值。

varnoold/varoattno：通常和 varno/varattno 相同，但是在等价变化的过程中，varno/varattno 的值可能发生变化，而 varnoold/varoattno 记录变化前的初值。

```
typedef struct Var
{
    Expr xpr;
    Index     varno;          /* 列属性所在的表在 Query->rtable 中的 rtindex*/
    AttrNumber varattno;      /* 列属性在表中的编号（第几列）*/
    Oid vartype;              /* 列属性对应的类型 */
    int32 vartypmod;          /* 列属性的精度（长度），Varchar(1024) */
    Oid varcollid;            /* OID of collation, or InvalidOid if none */
    Index     varlevelsup;    /* 列属性的相对位置，和子查询有关*/
    Index     varnoold;       /* varno 的初值 */
    AttrNumber varoattno;     /* varoattno 的初值 */
    int location;             /* 列属性出现在 SQL 语句中的位置 */
} Var;
```

例如有 SQL 语句：

```
SELECT st.sname FROM STUDENT st WHERE st.sno = ANY(SELECT sno FROM SCORE WHERE st.sno = sno);
```

其中 st.sno 虽然在子查询（SELECT sno FROM SCORE WHERE **st.sno = sno**）中被引用，但是它是父查询中的 STUDENT 表的属性，因此它的 varlevelsup 的值为 1，说明该属性所在的表在当前子查询的“上 1 层”，而子查询的约束条件中的另一个 sno 是 SCORE 表的列属性，它的 varlevelsup 的值为 0，意味着该列属性所在的表就在当前的层次，表 2-1 显示了 st.sno 对应的 Var 的各个值。

表 2-1　Var结构体成员变量的值

成员变量	描述
varno	1，父查询只有一个表STUDENT，所以处在Query->rtable链表的第一个
varattno	1，st.sno是STUDENT表的第1列

非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

续表

成员变量	描述
vartype	23，INT类型，参考pg_type.h
vartypmod	-1，无精度
varlevelsup	1，列属性是父查询表STUDENT的列属性

示例中另一个 SCORE.sno 对应的 Var 的成员变量的值如表 2-2 所示。

表 2-2　Var结构体成员变量的值

成员变量	描述
varno	1，子查询只有一个表SCORE，它在子查询树中的rtable链表中也是第1个
varattno	1，sno是SCORE表的第1列
vartype	23，INT类型，参考pg_type.h
vartypmod	-1，无精度
varlevelsup	0，列属性是当前层次查询表SCORE的列属性

示例中的 st.sname 对应的 Var 的成员变量的值如表 2-3 所示。

表 2-3　Var结构体成员变量的值

成员变量	描述
varno	1，父查询只有一个表STUDENT，所以处在Query->rtable链表的第一个
varattno	2，st.sname是STUDENT表的第2列
vartype	1043，Varchar类型，请参考pg_type.h
vartypmod	14，创建表时指定Varchar长度是10，这里的14包含了Header的长度
varlevelsup	0，列属性是当前层次查询表的列属性

2.3　RangeTblEntry 结构体

RangeTblEntry（范围表，简称 RTE）描述了查询中出现的表，它通常出现在查询语句的 FROM 子句中，范围表中既有常规意义上的堆表，还有子查询、连接表等。

```
typedef enum RTEKind
{
    RTE_RELATION,                /*  普通表 */
    RTE_SUBQUERY,                /*  子查询 */
    RTE_JOIN,                    /*  JOIN产生的表 */
```


非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
    RTE_FUNCTION,                   /*  可以作为表的函数返回的表 */
    RTE_TABLEFUNC,                  /*  TABLE 函数类型的表 */
    RTE_VALUES,                     /*  VALUES 表达式产生的表 */
    RTE_CTE,                        /*  WITH 语句附带的公共表 */
    RTE_NAMEDTUPLESTORE             /*  触发器使用 */
} RTEKind;
```

RangeTblEntry 根据成员变量 rtekind 来确定自己的类型，类型不同 RangeTblEntry 成员变量的作用也不同，例如针对 RTE_RELATION 类型的 RangeTblEntry，成员变量 relid 和 relkind 就是有用的，而对于 RTE_SUBQUERY 类型的 RangeTblEntry，它是一个子查询衍生的 RangeTblEntry，本身没有 relid 和 relkind，因此 relid 和 relkind 这时就处于无用的状态，而这时有效的是 subquery 成员变量。

```
typedef struct RangeTblEntry
{
    NodeTag       type;
    RTEKind       rtekind;      /* 范围表的类型*/

    // For RTE_RELATION，普通的表
    Oid       relid;            /* 表的 OID，来自 PG_CLASS 系统表 */
    char      relkind;          /* 表的类型，来自 PG_CLASS 系统表 */

    // For RTE_SUBQUERY，子查询类型的表
    Query       *subquery;      /* 子查询的查询树 */

    // For RTE_JOIN，连接类型的表
    JoinType jointype;          /* JOIN（连接）的类型 */
    List   *joinaliasvars;      /* JOIN（连接）的表的所有列的集合 */

    ......
} RangeTblEntry;
```

例如有 SQL 语句：

```
SELECT * FROM STUDENT LEFT JOIN SCORE ON TRUE, (SELECT * FROM TEACHER) AS t, COURSE, (VALUES(1,1))
AS NUM(x, y), GENERATE_SERIES(1,10) AS GS(z);
```

它对应的各种类型的 RangeTblEntry 如表 2-4 所示。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

表 2-4　RangeTblEntry结构体类型对照表

类型	对应的语句部分
RTE_RELATION	COURSE
RTE_JOIN	STUDENT LEFT JOIN SCORE ON TRUE
RTE_SUBQUERY	(SELECT * FROM TEACHER)
RTE_FUNCTION	GENERATE_SERIES(1,10) AS GS(z)
RTE_VALUES	(VALUES(1,2)) AS NUM(x, y)

示例中对应的 RangeTblEntry 在 Query->rtable 中的结构如图 2-1 所示，其中 RTE_JOIN 类型的 RangeTblEntry 中保存的是 STUDENT 和 SCORE 表中的投影的 Var，varno == 1 代表是 STUDENT 表中的列，varno == 2 代表是 SCORE 表中的列。

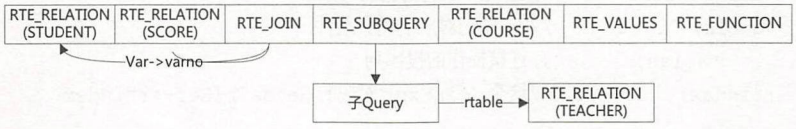


图 2-1　Query->rtable 链表内容示意图

2.4　RangeTblRef 结构体

RangeTblEntry 只保留在查询树的 Query->rtable 链表中，而链表是一个线性结构，它如何保存树状的关系代数表达式中的连接操作呢？答案是在 Query->jointree 中保存各个范围表之间的连接关系。

如果在 Query->jointree 中还保存同样的一份 RangeTblEntry，那么一方面会造成存储的冗余，另一方面也容易产生数据不一致的问题，因此在查询树的其他任何地方都不再存放新的 RangeTblEntry，每个范围表在 Query->rtable 链表中有且只能有一个，在其他地方使用到范围表都使用 RangeTblRef 来代替。RangeTblRef 是对 RangeTblEntry 的引用，因为 RangeTblEntry 在 Query->rtable 中的位置是确定的，因此可以用它在 Query->rtable 链表中的位置 rtindex 来标识。

```
typedef struct RangeTblRef
{
    NodeTag       type;
    int           rtindex;  //对应的 RangeTblEntry 在 Query->rtable 中的位置
} RangeTblRef;
```


严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

2.5 JoinExpr 结构体

在查询语句中如果显式地指定两个表之间的连接关系，例如 A LEFT JOIN B ON Pab 这种形式，就需要一个 JoinExpr 结构体来表示它们，我们先看一下 JoinExpr 结构体的定义。

```
typedef struct JoinExpr
{
    NodeTag  type;
    JoinType jointype;          //连接操作的类型，例如可以是 LeftJoin 等
    bool     isNatural;         //是否是自然连接
    Node     *larg;             //连接操作的 LHS（左侧）的表
    Node     *rarg;             //连接操作的 RHS（右侧）的表
    List     *usingClause;      //using 子句对应的约束条件
    Node     *quals;            //on 子句对应的约束条件
    Alias        *alias;        //连接操作的投影列
    int      rtindex;           //这个 JoinExpr 对应的 RangeTblRef->rtindex
} JoinExpr;
```

我们来看几个示例。

例 1：SELECT * FROM STUDENT INNER JOIN SCORE ON STUDENT.sno = SCORE.sno; 这个示例语句明确指定了两个表进行内连接操作，因此它们需要用 JoinExpr 来表示，JoinExpr->quals 中保存的是 STUDENT.sno = SCORE.sno。

例 2：SELECT * FROM STUDENT LEFT JOIN SCORE ON STUDENT.sno = SCORE.sno; 这个示例语句中明确指定了两个表进行左外连接操作，因此它也必须使用 JoinExpr 来表示，JoinExpr->quals 中保存的是 STUDENT.sno = SCORE.sno。

例 3：SELECT * FROM STUDENT LEFT JOIN SCORE ON TRUE LEFT JOIN COURSE ON SCORE.cno = COURSE.cno; 这个示例语句中明确指定了 3 个表之间的连接关系，它需要有两个 JoinExpr 来表示，如图 2-2 所示。

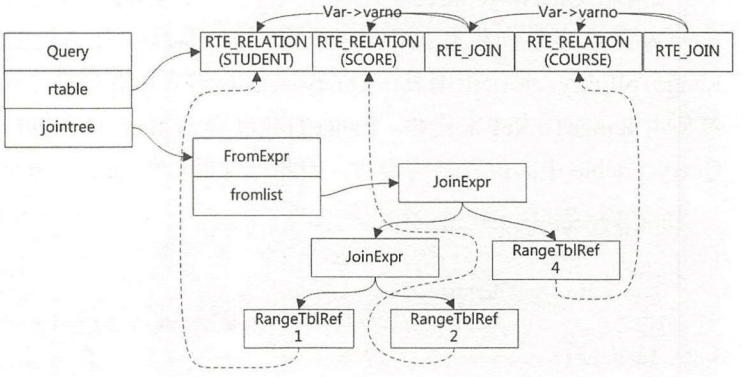


图 2-2 JoinExpr 的内存结构示意图

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

2.6　FromExpr 结构体

FromExpr 和 JoinExpr 是用来表示表之间的连接关系的结构体，通常来说，FromExpr 中的各个表之间的连接关系是 Inner Join，这样就可以在 FromExpr->fromlist 中保存任意多个表，默认它们之间是内连接的关系，我们先来看一下它的结构体的定义。

```
typedef struct FromExpr
{
    NodeTag  type;
    List    *fromlist; //FromExpr 中包含几个表
    Node    *quals;    //fromlist 中的表之间的约束条件
} FromExpr;
```

我们来看几个示例。

例 1： SELECT * FROM STUDENT, SCORE, COURSE WHERE STUDENT.sno = SCORE.sno；这个示例中的 3 个表没有明确地指定连接关系，默认它们是 InnerJoin，因此可以通过 FromExpr 来表示，FromExpr->fromlist 中保存了语句中的 3 个表，FromExpr->quals 中保存了 STUDENT.sno = SCORE.sno 这个约束条件。

例 2： SELECT * FROM STUDENT, SCORE LEFT JOIN COURSE ON SCORE.cno = COURSE.cno；如图 2-3 所示。

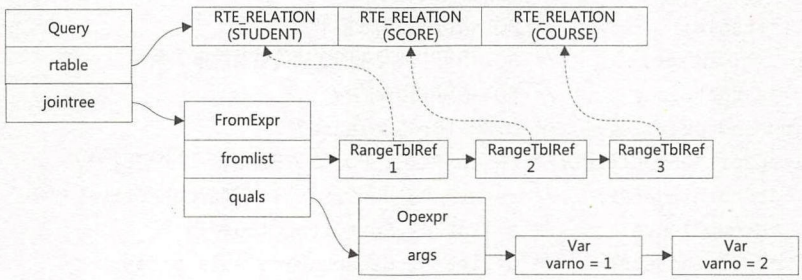


图 2-3　FromExpr 的内存结构示意图

2.7　Query 结构体

Query 结构体是查询优化模块的输入参数，其源自于语法分析模块，一个 SQL 语句在执行过程中，经过词法分析、语法分析和语义分析之后，会生成一棵查询树，PostgreSQL 用 Query 结构体来表示查询树。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

查询优化模块在获取到查询树之后，开始对查询树进行逻辑优化，也就是对查询树进行等价变换，将其重写成一棵新的查询树，这个新的查询树又作为物理优化的输入参数，进行物理优化。

```
typedef struct Query
{
    NodeTag  type; /* Node类型 */
    CmdType  commandType;      /* 语句类型：select|insert|update|delete|utility */
    QuerySource querySource;   /* where did I come from? */
    uint32   queryId;          /* query identifier (can be set by plugins) */
    bool     canSetTag;        /* do I set the command result tag? */
    Node     *utilityStmt;     /* utility类型的语句，通常为DDL语句*/
    int      resultRelation;   /* INSERT/UPDATE/DELETE操作的目标表 */
    bool     hasAggs;          /* 是否在tlist 或 havingQual上有聚集操作 */
    bool     hasWindowFuncs;   /* 是否有窗口函数*/
    bool     hasTargetSRFs;    /* 是否有SRF函数 */
    bool     hasSubLinks;      /* 是否包含SubLink或SubQuery */
    bool     hasDistinctOn;    /* distinctClause is from DISTINCT ON */
    bool     hasRecursive;     /* WITH RECURSIVE was specified */
    bool     hasModifyingCTE;  /* has INSERT/UPDATE/DELETE in WITH */
    bool     hasForUpdate;     /* FOR [KEY] UPDATE/SHARE was specified */
    bool     hasRowSecurity;   /* rewriter has applied some RLS policy */
    List     *cteList;         /* 通用表达式子句 */
    List     *rtable;          /* SQL语句涉及的表清单 */
    FromExpr  *jointree;       /* SQL语句中涉及表的连接关系及约束关系*/
    List     *targetList;      /* SQL语句中涉及的*/
    OverridingKind override;   /* OVERRIDING clause */
    OnConflictExpr *onConflict;/* ON CONFLICT DO [NOTHING | UPDATE] */
    List     *returningList;   /* return-values list (of TargetEntry) */
    List     *groupClause;     /* a list of SortGroupClause's */
    List     *groupingSets;    /* a list of GroupingSet's if present */
    Node     *havingQual;      /* having子句 */
    List     *windowClause;    /* a list of WindowClause's */
    List     *distinctClause;  /* a list of SortGroupClause's */
    List     *sortClause;      /* a list of SortGroupClause's */
    Node     *limitOffset;     /* # of result tuples to skip (int8 expr) */
    Node     *limitCount;      /* # of result tuples to return (int8 expr) */
    List     *rowMarks;        /* a list of RowMarkClause's */
    Node     *setOperations;   /* set-operation tree if this is top level of
                                * a UNION/INTERSECT/EXCEPT query */
    List     *constraintDeps;  /* a list of pg_constraint OIDs that the query
```

版权所有，严禁传播，违者自负法律责任！


```
                                * depends on to be semantically valid */
    List     *withCheckOptions;/* a list of WithCheckOption's, which are
                                * only added during rewrite and therefore
                                * are not written out as part of Query. */

    /*
     * The following two fields identify the portion of the source text string
     * containing this query.  They are typically only populated in top-level
     * Queries, not in sub-queries.  When not set, they might both be zero, or
     * both be -1 meaning "unknown".
     */
    int          stmt_location;     /* start location, or -1 if unknown */
    int          stmt_len;          /* length in bytes; 0 means "rest of string" */
} Query;
```

rtable：在查询中 FROM 子句后面会指出需要进行查询的范围表，可能是对单个范围表进行查询，也可能是对几个范围表做连接操作，rtable 中则记录了这些范围表。rtable 是一个 List 指针，所有要查询的范围表就记录在这个 List 中，每个表以 RangeTblEntry 结构体来表示，因此 rtable 是一个以 RangeTblEntry 为节点的 List 链表。

jointree：rtable 中列出了查询语句中的表，但没有明确指出各个表之间的连接关系，这个连接的关系则通过 jointree 来标明，jointree 是一个 FromExpr 类型的结构体，它有 3 种类型的节点：FromExpr、JoinExpr 和 RangeTblRef。

targetlist：targetlist 中包含了需要投影（Project）的列，也就是 SFW 查询中的投影列。

例 1：SELECT * FROM STUDENT WHERE SNO = 1; Query 结构体中变量对应的值如表 2-5 所示。

表 2-5　例 1 对应的Query结构体内容

成员变量	描述
rtable	rtable链表中只有一个RangeTblEntry节点来表示STUDENT表
jointree	FromExpr: fromlist：有一个RangeTblRef节点来表示STUDENT表 FromExpr: quals：有一个OPExpr（操作符表达式）表示SNO = 1的操作
targetlist	投影列*号被展开，里面是TargetEntry类型的3个节点，分别代表STUDENT表的3个列（sno, sname, ssex）

例 2：SELECT st.sname, sc.degree FROM STUDENT st, SCORE sc WHERE st.sno = sc.sno; Query 结构体变量对应的值如表 2-6 所示。

版权所有，严禁传播，违者自负法律责任！

表 2-6　例 2 对应的Query结构体内容

成员变量	描述
rtable	rtable链表中有两个RangeTblEntry节点来表示STUDENT和SCORE两个表
jointree	FromExpr:fromlist：有两个RangeTblRef节点表示STUDENT和SCORE表 FromExpr:quals：一个OPExpr（操作符表达式）表示st.sno = sc.sno
targetlist	语句指定了两个投影列，targetlist链表中有两个TargetEntry，分别代表STUDENT表中的sno和SCORE表中的degree

例 3：SELECT st.sname, sc.degree FROM STUDENT st INNER JOIN SCORE sc ON st.sno = sc.sno; Query 结构体变量对应的值如表 2-7 所示。

表 2-7　例 3 对应的Query结构体内容

成员变量	描述
rtable	rtable链表中有3个RangeTblEntry节点，前两个表示STUDENT和SCORE两个表，第3个表示STUDENT和SCORE的连接关系
jointree	FromExpr: fromlist：有一个JoinExpr类型的节点 JoinExpr: larg：一个RangeTblRef表示STUDENT表 JoinExpr: rarg：一个RangeTblRef表示SCORE表 JoinExpr: quals：一个OPExpr（操作符表达式）表示st.sno = sc.sno FromExpr:quals：NULL
targetlist	语句指定了两个投影列，targetlist链表中有两个TargetEntry，分别代表STUDENT表中的sname和SCORE表中的degree

例 4：SELECT st.sname, c.cname, sc.degree FROM STUDENT st , COURSE c INNER JOIN SCORE sc ON c.cno = sc.cno WHERE st.sno = sc.sno;如图 2-4 所示。

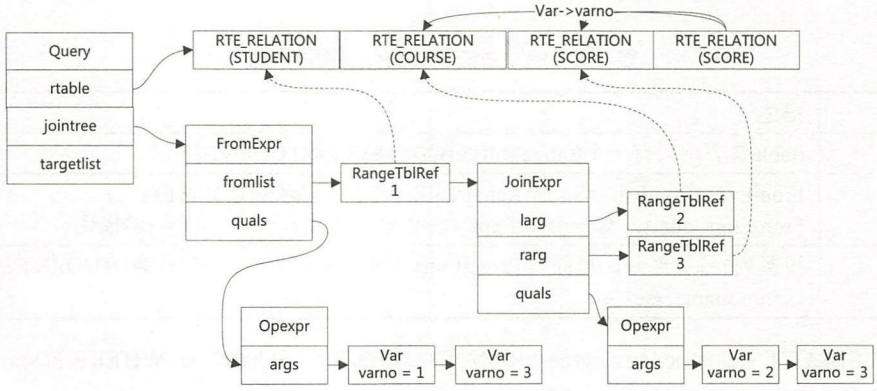


图 2-4　例 4 对应的 Query 结构体内存示意图

版权所有，严禁传播，违者自负法律责任！

2.8　查询树的展示

PostgreSQL 数据库提供了参数让我们查看查询树和执行计划树，如表 2-8 所示。

表 2-8　展示查询树的GUC参数

参数名称	默认值	描述
debug_pretty_print	On	打开该参数在打印查询树和执行计划时会以结构化的方式来展示，便于对查询树进行分析
debug_print_parse	Off	打开该参数可以打印查询树
debug_print_rewritten	Off	打开该参数可以打印重写（视图）之后的查询树
debug_print_plan	Off	打开该参数可以打印执行计划

在调试查询优化源代码的过程中，通常需要不止一次地打印查询树，因为在逻辑优化阶段需要对查询树进行重写，我们可以通过打印查询树的功能来查看查询树重写前和重写后的区别，读者可以在查询重写前和重写后在源代码中增加对应的函数（例如 elog_node_display 函数），这样能更方便地查看查询重写的内容。例如我们可以在子查询提升函数的前后分别通过 elog_node_display 函数来打印查询树，这样就能看出子查询提升前和提升后的查询树的内容发生了哪些变化。

增加 elog_node_display 函数到代码中这种方法在我们尝试对查询优化功能进行修改的时候同样有用，例如现在要在逻辑优化中增加一个新的优化规则，我们可以通过打印查询树更好地规划修改方案。

2.9　查询树的遍历

在对查询树进行重写的过程中，需要经常性地对查询树（Query）进行遍历，从中查找、替换、编辑某个结构体的值，以实现重写的功能。例如在子查询提升的过程中，由于在父查询和子查询中的 Var 具有不同的层次关系，因此它们的 varlevelsup 是不同的，但是在子查询提升之后，这种层次关系就消失了，因此需要调整一些 Var 的 varlevelsup 的值，这时候就需要从查询树中找出这些 Var 并编辑或替换这些 Var。

由于在 PostgreSQL 数据库中所有的“节点”都是以类似于“类”的方式实现的，因此我们可以由基类 Node 的指针代表任何节点，并且能通过 Nodetag 快速地识别出当前 Node 的真实类型，这就为我们遍历查询树提供了便利。PostgreSQL 数据库通过提供 query_tree_mutator 函数和

版权所有，严禁传播，违者自负法律责任！

query_tree_walker 函数来对查询树进行遍历，实际上还需要借助 expression_tree_mutator 函数和 expression_tree_walker 函数来实现，walker 函数的主要作用是对查询树进行遍历并且可能会修改某个结构体中的值，但是它不增加或删除节点，如果要增加或删除查询树中的某个节点，则应该使用 mutator 函数。这部分的源代码主要是借用递归的思想遍历各种表达式的结构体，有兴趣的读者可以自行分析。

2.10 执行计划的展示

本书中大量地使用 EXPLAIN 语句来展示查询语句的执行计划，这个执行计划是一个非完全的二叉树，每个父节点至少有一个子节点，同时最多有两个子节点，PostgreSQL 数据库的查询执行器通过对这个二叉树迭代执行来获得查询结果，EXPLAIN 的语法定义如下：

```
EXPLAIN [ ( _option_ [, ...] ) ] _statement_
EXPLAIN [ ANALYZE ] [ VERBOSE ] _statement_
```

其中的_option_可以是如下内容：

```
ANALYZE [ _boolean_ ]
VERBOSE [ _boolean_ ]
COSTS [ _boolean_ ]
BUFFERS [ _boolean_ ]
TIMING [ _boolean_ ]
FORMAT { TEXT | XML | JSON | YAML }
```

查询优化的主要目的就是将一个查询树（关系代数）转换为一个代价最低的非完全二叉树的执行算式，算式中的每一个节点对应一个物理路径，EXPLAIN 语句打印的就是这个非完全二叉树，例如：

```
postgres=# EXPLAIN SELECT * FROM STUDENT, (SELECT * FROM SCORE) as sc;
                        QUERY PLAN
-------------------------------------------------------------
 Nested Loop  (cost=0.00..2.15 rows=7 width=31)
   ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
   ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=19)
(3 rows)
```

这个二叉树中有一个父节点和两个子节点，子节点分别对 SCORE 和 STUDENT 进行顺序扫描，父节点是对子节点的扫描结果做 NestloopJoin，cost=0.00..2.15 中的 0.00 代表的是启动代价，2.15 代表的是这个节点和它的子节点共同的整体代价（启动代价和整体代价可以参考第 6 章）。

版权所有，严禁传播，违者自负法律责任！

EXPLAIN 还带有一些关键字作为选项，它们分别的含义如表 2-9 所示。

表 2-9　EXPLAIN的参数

关键字名	说明
VERBOSE	指定了VERBOSE关键字之后，打印的信息会更丰富
ANALYZE	如果指定了ANALYZE，不但打印查询优化模块估算的代价，还会打印出语句执行的实际代价，注意：增加了ANALYZE关键字之后，查询语句会真正地执行
BUFFERS	需要和ANALYZE同时使用，打印缓冲区的命中率
COSTS	PostgreSQL数据库默认打印代价，可以通过（COSTS OFF）关闭代价的打印，本选项默认是打开的
TIMING	需要和ANALYZE同时使用，ANALYZE默认会统计各个节点的实际运行时间，通过TIMING OFF，可以让ANALYZE不统计时间，只统计处理的元组的数量

2.11　小结

本章主要介绍了查询树的构造形态，读者可以了解到查询树中最重要的几个成员变量，例如 Var、RangeTblEntry、FromExpr、JoinExpr 等，这样在针对查询树应用优化规则的时候，用户才能清楚地了解查询树变换的真正意图。

另外，查询优化的输入参数不只有查询树对应的结构体 Query，另外还有 PlannerInfo 结构体（包括 PlannerGlobal 结构体），这个结构体是查询优化的上下文信息，它贯穿在整个查询优化的过程之中，因此现阶段很难清楚地说明它的每个成员变量的具体作用，因此我们把对 PlannerInfo 结构体的说明“散落”在本书的各个章节，在用到 PlannerInfo 的每个成员变量的时候再对其进行说明。

版权所有，严禁传播，违者自负法律责任！

3 第3章 逻辑重写优化

依照 PostgreSQL 数据库逻辑优化的源代码的分布情况，我们把逻辑优化分成了两部分：逻辑重写优化和逻辑分解优化，本章重点开始分析逻辑重写优化部分的源代码。划分的依据是：在逻辑重写优化阶段主要还是对查询树进行“重写”，也就是说在查询树上进行改造，改造之后还是一颗查询树，而在逻辑分解阶段，会将查询树打散，会重新建立等价于查询树的逻辑关系，逻辑重写优化的函数关系如图 3-1 所示。

版权所有，严禁传播，违者自负法律责任！

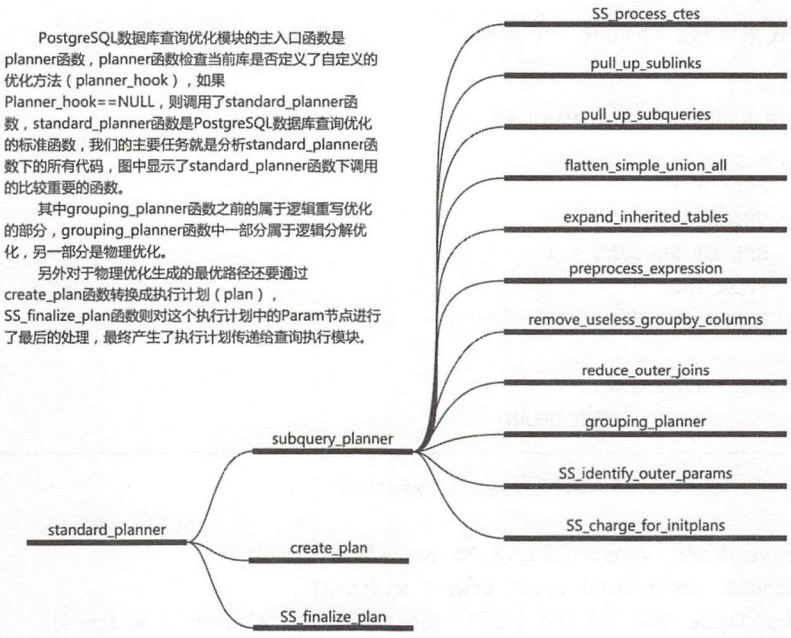


图 3-1　逻辑重写优化函数调用图

3.1　通用表达式

所谓通用表达式对应的是 WITH 语句，它的作用和子查询类似，都是一个单独的逻辑操作，但是数据库不对通用表达式做提升优化，因为使用通用表达式方式来实现的子表通常会多次被使用，也就是说它具有“一次求值，多次使用”的特点，下面看一个简单的通用表达式的例子：

```
postgres=# EXPLAIN
postgres-# WITH CTE AS
postgres-# (Select * FROM STUDENT)
postgres-# SELECT sname FROM CTE;
                      QUERY PLAN
-------------------------------------------------------------
 CTE Scan on cte  (cost=1.07..1.21 rows=7 width=38)
   CTE cte
     -> Seq Scan on student  (cost=0.00..1.07 rows=7 width=19)
(3 rows)
```

另外 PostgreSQL 数据库还支持了递归通用表达式的功能，在表达式中通过“初值 UNION

版权所有，严禁传播，违者自负法律责任！

递归值”的方式来实现，下面看一个示例：

```
postgres=# EXPLAIN
postgres-# WITH RECURSIVE INC(VAL) AS
postgres-# (
postgres-#     SELECT 1
postgres-#     UNION ALL
postgres-#     SELECT INC.VAL + 1
postgres-#     FROM INC
postgres-#     WHERE INC.VAL < 10
postgres-# )
postgres-# SELECT * FROM INC;
                          QUERY PLAN
------------------------------------------------------------------------------
 CTE Scan on inc  (cost=2.95..3.57 rows=31 width=4)
   CTE inc
     ->  Recursive Union  (cost=0.00..2.95 rows=31 width=4)
           ->  Result  (cost=0.00..0.01 rows=1 width=4)
           ->  WorkTable Scan on inc inc_1  (cost=0.00..0.23 rows=3 width=4)
                 Filter: (val < 10)
(6 rows)
```

由于通用表达式的处理类似于一个子查询，它通过给子查询生成子 Plan 的方式来实现，生成子 Plan 的查询优化和父 Plan 的查询优化复用了同一份代码，因此这里就不再展开叙述了。

3.2 子查询提升

应用程序通过 SQL 语句来操作数据库时会使用大量的子查询，这种写法比直接对两个表做连接操作结构上要更清晰，尤其是在实现一些比较复杂的查询语句时，子查询由于具有一定的独立性，会使写出的 SQL 脚本更容易理解。

从大的方向上分类，子查询可以分为相关子查询和非相关子查询。

相关子查询：指在子查询语句中引用了外层表的列属性，这就导致外层表每获得一个元组，子查询就需要重新执行一次；

非相关子查询：指在子查询语句是独立的，和外层的表没有直接的关联，子查询可以单独执行一次，外层表可以重复利用子查询的执行结果。

PostgreSQL 数据库还基于子查询所在的位置和作用的不同，将子查询细分成了两类，一类

版权所有，严禁传播，违者自负法律责任！

称为子连接（SubLink），另一类称为子查询（SubQuery）。那么如何来区分子查询和子连接呢？通常而言，如果它是以范围表的方式存在的，那么就称为子查询，例如下面的示例语句中是以一个范围表的方式存在的。

```
postgres=# EXPLAIN SELECT * FROM STUDENT, (SELECT * FROM SCORE) as sc;
                        QUERY PLAN
-------------------------------------------------------------
 Nested Loop  (cost=0.00..2.15 rows=7 width=31)
   ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
   ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=19)
(3 rows)
```

如果它以表达式的方式存在，那么就称为子连接，例如下面的标量子查询，它是一个 EXPR_SUBLINK 类型的子连接，它是投影中的一个表达式。

```
postgres=# EXPLAIN SELECT (SELECT AVG(degree) FROM SCORE), sname FROM STUDENT;
                        QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on student  (cost=1.02..2.09 rows=7 width=43)
   InitPlan 1 (returns $0)
     ->  Aggregate  (cost=1.01..1.02 rows=1 width=32)
           ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=4)
(4 rows)
```

在实际应用中可以通过子句所处的位置来区分子连接和子查询，出现在 FROM 关键字后的子句是子查询语句，出现在 WHERE/ON 等约束条件中或投影中的子句是子连接语句。

子查询固然从语句的逻辑层次上是清晰的，但是使用的方法不同，它的效率有高有低，通常而言，相关子查询是值得提升的，因为其执行结果和父查询相关，也就是说父查询的每一条元组都对应着子查询的重新求值，而非相关子查询则可以不提升，因为可以一次求值多次使用，但是在实际应用中还需要根据具体的情况做具体的分析。

3.2.1　提升子连接

子连接（SubLink）是子查询的一种特殊情况，由于子连接出现在 WHERE/ON 等约束条件中，因此它通常伴随着 ANY/ALL/IN/EXISTS/SOME 等谓词同时出现，PostgreSQL 数据库依据每种不同的谓词区分 SUBLINK 的类型。

```
typedef enum SubLinkType
{
    EXISTS_SUBLINK,     // [NOT] EXISTS 谓词
```

版权所有，严禁传播，违者自负法律责任！


```
    ALL_SUBLINK,        //ALL 谓词
    ANY_SUBLINK,        // ANY/IN/SOME 谓词
    ......
} SubLinkType;
```

PostgreSQL 数据库主要对 ANY_SUBLINK 和 EXISTS_SUBLINK 两种类型的子连接尝试进行提升，因此下面我们主要分析这两种类型的子连接，通过分析 gram.y 文件（语法分析文件），发现主要是 IN/ANY 谓词生成 ANY_SUBLINK，EXISTS 谓词生成 EXISTS_SUBLINK，如表 3-1 所示。

表 3-1　子连接语法形式和逻辑形式对照表

谓词	形式	描述
[NOT] IN	LH [NOT] IN EXPR	如果提升，则变为[反]半连接，即[Anti-] Semi Join）
ANY/SOME	LH OP ANY EXPR	如果提升，则变为半连接，即Semi Join
[NOT] EXISTS	[NOT] EXISTS EXPR	如果提升，则变为[反]半连接，即[Anti-] Semi Join）

下面通过一些例子来感性地看一下哪些子连接能够提升。例如下面的 SQL 语句，它是一个 EXISTS 类型的相关子连接，由执行计划可以看出，子连接已经被提升了，提升之后的逻辑运算符是 SemiJoin，物理运算符是 HashJoin，提升之后通过将内表 Hash 化，降低了算法的复杂度，避免了重复多次执行子查询的问题。

```
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE EXISTS (SELECT a FROM TEST_B WHERE TEST_A.a
= TEST_B.a);
                        QUERY PLAN
-----------------------------------------------------------------------
 Hash Semi Join  (cost=54.05..112.30 rows=2 width=16)
   Hash Cond: (test_a.a = test_b.a)
   ->  Seq Scan on test_a  (cost=0.00..57.56 rows=256 width=16)
   ->  Hash  (cost=54.02..54.02 rows=2 width=4)
         ->  Seq Scan on test_b  (cost=0.00..54.02 rows=2 width=4)
(5 rows)
```

再看一个 EXISTS 类型的非相关子连接的例子。从执行计划可以看出，子连接形成了一个单独的子执行计划，查询执行器会对这个子执行计划进行单独求解，求解的结果则作为一个 One-Time Filter 来决定是否对 TEST_A 表进行扫描，这里虽然没有做到“一次求解多次使用”，但是做到了“一次求解决定全局”。

```
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE EXISTS (SELECT a FROM TEST_B);
                        QUERY PLAN
-----------------------------------------------------------------------
 Result  (cost=27.01..84.57 rows=256 width=16)
```

版权所有，严禁传播，违者自负法律责任！


```
  One-Time Filter: $0
  InitPlan 1 (returns $0)
    -> Seq Scan on test_b  (cost=0.00..54.02 rows=2 width=0)
  -> Seq Scan on test_a  (cost=27.01..84.57 rows=256 width=16)
(5 rows)
```

由此可以记住一个大的原则：**EXISTS 类型的相关子连接会被提升，非相关子连接会形成子执行计划单独求解**。需要注意的是，还需要保证 EXIST 类型的相关子连接是“简单”的才能被提升，例如在下面的示例中，由于投影中含有聚集函数，导致了这个 EXISTS 类型的相关子连接也不能提升。

```
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE EXISTS (SELECT SUM(a) FROM TEST_B WHERE TEST_A.a
= TEST_B.a);
                        QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on test_a  (cost=0.00..13891.80 rows=128 width=16)
   Filter: (SubPlan 1)
   SubPlan 1
     -> Aggregate  (cost=54.03..54.04 rows=1 width=8)
          -> Seq Scan on test_b  (cost=0.00..54.02 rows=2 width=4)
               Filter: (test_a.a = a)
(6 rows)
```

如下面的示例所示，PostgreSQL 数据库对 ANY 类型的非相关子连接进行了提升， 虽然这个 SQL 语句看上去是非相关子连接，但是它的约束条件中的 a 作为‘>’操作符的左值，和这个子连接有很大的关系，也就是说它可以形成一个相关的约束条件，从执行计划可以看出，提升之后它们有了一个约束条件 TEST_A.a > TEST_B.a，也就是说 ANY 类型的子连接天然带有“相关”的特性。另外从执行计划也可以看出，子连接被提升之后变成了对 TEST_B 表的扫描，并且对扫描之后的表进行了物化，物化的主要作用就是“一次扫描，多次使用”，因此这种提升是可以带来收益的。

```
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE a > ANY (SELECT a FROM TEST_B);
                      QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop Semi Join  (cost=0.00..118.41 rows=85 width=16)
   Join Filter: (test_a.a > test_b.a)
   -> Seq Scan on test_a  (cost=0.00..57.56 rows=256 width=16)
   -> Materialize  (cost=0.00..54.03 rows=2 width=4)
        -> Seq Scan on test_b  (cost=0.00..54.02 rows=2 width=4)
(5 rows)
```

版权所有，严禁传播，违者自负法律责任！

PostgreSQL 数据库对 ANY 类型的相关子连接没有进行提升。从下面的示例可以看出，子连接形成了一个单独的子执行计划。

```
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE a > ANY (SELECT a FROM TEST_B WHERE TEST_A.b
= TEST_B.b);
                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on test_a  (cost=0.00..6974.04 rows=128 width=16)
   Filter: (SubPlan 1)
   SubPlan 1
     -> Seq Scan on test_b  (cost=0.00..54.02 rows=2 width=4)
          Filter: (test_a.b = b)
(5 rows)
```

这里需要说明的是，实际上面示例的相关子连接应该也是能够提升的，例如 Oracle 数据库对这种情况就进行了提升，它的执行计划如下：

```
SQL> explain plan for SELECT * FROM TEST_A WHERE a > ANY (SELECT a FROM TEST_B WHERE TEST_A.b
= TEST_B.b);
| Id  | Operation           | Name    | Rows  | Bytes | Cost (%CPU)| Time
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |         |     1 |    26 |     4   (0)| 00:00:01
|*  1 |  HASH JOIN SEMI     |         |     1 |    26 |     4   (0)| 00:00:01
|   2 |   TABLE ACCESS FULL | STUDENT |     1 |    13 |     2   (0)| 00:00:01
|   3 |   TABLE ACCESS FULL | SCORE   |     1 |    13 |     2   (0)| 00:00:01

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("TEST_A"."B"="TEST_B"."B")
       filter("A">"A")
```

PostgreSQL 没有将这种子连接提升是由于 ANY 类型的子连接提升分成了两个步骤，第一个步骤是将子连接提升成子查询，第二个步骤则进行子查询提升，对于 ANY 类型的相关子连接在提升成子查询之后会形成如下形式的语句：

```
SELECT * FROM TEST_A SEMI JOIN (SELECT a FROM TEST_B WHERE TEST_A.b = TEST_B.b) AS b WHERE
TEST_A.a > TEST_B.a;
```

这个 SQL 语句中的子查询中引用了父查询的属性 TEST_A.b，这种情况 PostgreSQL 数据库在语法上是不支持的，自然在源代码中也没有支持这种逻辑。当然，PostgreSQL 数据库在 9.3 版本之后支持了 Lateral 语义，可以通过 Lateral 关键字来支持子查询引用父查询的属性，但

版权所有，严禁传播，违者自负法律责任！

PostgreSQL 数据库目前还没有支持这种 ANY 类型的相关子连接的提升。

由此对于 ANY 类型的子连接我们也可以得到一个感性的原则：ANY 类型的非相关子连接可以提升（仍然需要是“简单”的子连接），并且可以通过物化的方式进行优化，而 ANY 类型的相关子连接目前还不能提升。

PostgreSQL 数据库为子连接定义了单独的结构体——SubLink 结构体，其中主要描述了子连接的类型、子连接的操作数及操作符等信息。

```
typedef struct SubLink
{
    Expr  xpr;
    SubLinkType subLinkType;    /* 子连接类型*/
    int       subLinkId;        /* 编号 */
    Node      *testexpr;        /* 针对不同谓词的操作 */
    List      *operName;        /* 子连接的操作符*/
    Node      *subselect;       /* 子连接中的子句所产生的查询树 */
    int       location;         /* 关键字在 SQL 语句中的位置 */
} SubLink;
```

下面看一个具体的例子。对于语句 SELECT sname FROM STUDENT WHERE sno > ANY (SELECT sno FROM SCORE)，它的子连接语句是 sno > ANY(SELECT sno FROM SCORE)，对应到 SubLink 结构体中的每个变量的情况如表 3-2 所示。

表 3-2　子连接内存结构示例

变量	说明
subLinkType	ANY谓词产生的ANY_SUBLINK
subLinkId	语句中只有一个子连接，编号为1
testexpr	操作符表达式，左操作数是Var，代表STUDENT.sno，操作符是“>”，右操作数类型是Param，代表SCORE.sno，也就是子连接中的投影列（targetlist）
operName	操作符是“>”
subselect	子连接中的子句生成的查询树（Query），这里是SQL语句中的“SELECT sno FROM SCORE”生成的查询树

3.2.1.1　pull_up_sublinks 函数

如图 3-2 所示，子连接提升的主要过程是在 pull_up_sublinks 函数中实现的，该函数的主要参数是 Query->jointree，Query->jointree 的说明可以查阅本书关于 Query 结构体的说明。

版权所有，严禁传播，违者自负法律责任！

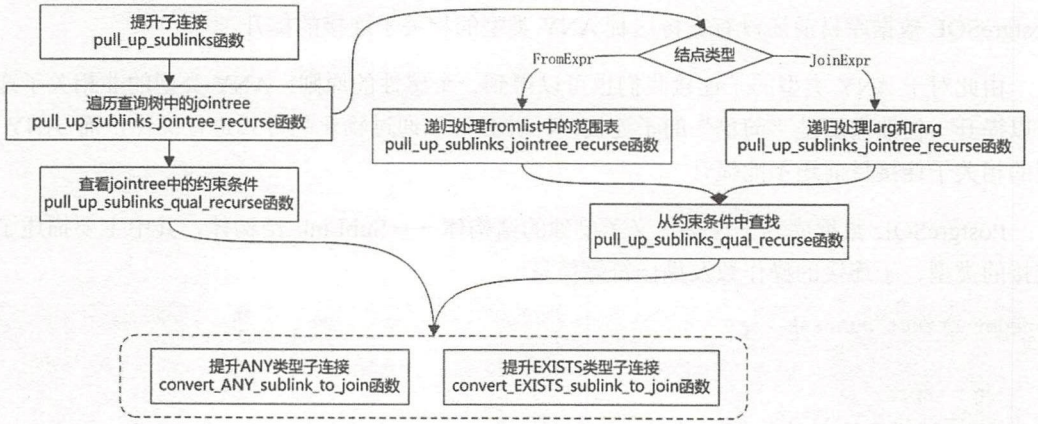


图 3-2　子连接函数调用关系

3.2.1.2　pull_up_sublinks_qual_recurse 函数

pull_up_sublinks 函数又调用 pull_up_sublinks_qual_recurse 函数，它的输入参数如表 3-3 所示。

表 3-3　pull_up_sublinks_qual_recurse函数参数说明

参数名	参数类型	描述
root	[IN] PlannerInfo *	查询优化模块的上下文信息结构体
jtnode	[IN] Node *	需要递归处理的节点，可能是RangeTblRef、FromExpr或JoinExpr
relids	[OUT] Relids *	输出参数，jtnode参数中涉及的表的集合
返回值	Node *	经过子查询提升处理之后的jtnode节点

pull_up_sublinks_qual_recurse 函数主要是对 Query->jointree 进行递归分析，Query->jointree 节点分为三种类型：RangeTblRef、FromExpr 或 JoinExpr，针对这 3 种类型的节点 pull_up_sublinks_qual_recurse 函数做了不同的处理。

1）RangeTblRef：RangeTblRef 一定是 Query->jointree 上的叶子节点，因此是递归的结束条件，保存当前表的 relid 返回给上层，执行到这个分支通常有两种情况。

a) 情况一：查询语句只有单表，没有连接操作，这种情况递归处理结束，另外去查看子连接是否满足提升的其他条件。

b) 情况二：查询语句有连接关系，在对 FromExpr->fromlist、JoinExpr->larg 或者 JoinExpr->rarg 递归的过程中，遍历到了叶子节点 RangeTblRef，这时候需要将这个

版权所有，严禁传播，违者自负法律责任！

RangeTblRef 节点的 relids 返回给上一层，主要用于判断该子查询能否被提升，例如子连接的左操作数如果是左连接的 LHS 的一个列属性，则这个子连接就不能提升，具体可以参考下面 JoinExpr 的说明。

2）FromExpr：

a) 对 FromExpr->fromlist 中的节点做递归遍历，对每个节点递归调用 pull_up_sublinks_jointree_recurse 函数，一直处理到叶子节点 RangeTblRef 才返回。

b) 调用 pull_up_sublinks_qual_recurse 函数处理 FromExpr->qual，对其中可能出现的 ANY_SUBLINK 或 EXISTS_SUBLINK 做处理。

3）JoinExpr：

a) 分别调用 pull_up_sublinks_jointree_recurse 函数递归处理 JoinExpr->larg 和 JoinExpr->rarg，一直处理到叶子节点 RangeTblRef 才返回。另外还需要根据连接操作的类型区分子连接是否能够被提升，以左连接为例，如果子连接的左操作数是 LHS 的列属性，则该子连接不能被提升，因为提升后两个查询树不等价。例如对于如下 SQL 语句，子连接是不能提升的。

```
postgres=# EXPLAIN SELECT sno FROM STUDENT LEFT JOIN COURSE ON STUDENT.sno >
ANY(SELECT sno FROM SCORE);
                          QUERY PLAN
---------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..176085.32 rows=3850 width=4)
   Join Filter: (SubPlan 1)
   -> Seq Scan on student  (cost=0.00..1.07 rows=7 width=4)
   -> Materialize  (cost=0.00..26.50 rows=1100 width=0)
        -> Seq Scan on course  (cost=0.00..21.00 rows=1100 width=0)
   SubPlan 1
     -> Materialize  (cost=0.00..40.60 rows=2040 width=4)
          -> Seq Scan on score  (cost=0.00..30.40 rows=2040 width=4)
(8 rows)
```

我们用“反证法”来检验一下这种情况，假设能够将上面示例中的子连接提升，转换成可以执行的 SQL 语句如下：

```
SELECT sno FROM (SELECT sno FROM STUDENT WHERE sno > ANY (SELECT sno FROM SCORE))
sc LEFT JOIN COURSE ON TRUE;
```

显然，提升前和提升后的两个语句的执行结果是不等价的，提升前的 SQL 语句中 STUDENT 表的所有元组都会出现在查询结果中，而“提升后”的 SQL 语句中的

版权所有，严禁传播，违者自负法律责任！

STUDENT 表中的有些元组可能会被过滤掉，这就导致了结果不同。下面将逻辑连接操作符和子连接提升的关系总结在表 3-4 中。

表 3-4　逻辑连接操作符和子连接提升的关系

连接类型	提升情况
内连接	连接的左操作数可以是LHS或者RHS的列属性，均可提升
左连接	左操作数只能是RHS的列属性子连接才能提升
右连接	左操作数只能是LHS的列属性子连接才能提升
全连接	子连接不能提升

b) 调用 pull_up_sublinks_qual_recurse 函数处理 JoinExpr->quals，对其中可能出现的 ANY_SUBLINK 或 EXISTS_SUBLINK 做处理，需要注意的是，根据连接类型的不同，pull_up_sublinks_qual_recurse 函数的 available_rels1 参数的输入值是不同的。

3.2.1.3　ANY 类型子连接的提升条件

ANY 类型的子连接的提升流程在 convert_ANY_sublink_to_join 函数中实现，该函数对查询树中的子连接进行了结构变换，将子连接变为和上层表的 SemiJoin，函数的参数说明如表 3-5 所示。

表 3-5　convert_ANY_sublink_to_join函数的参数说明

参数名	参数类型	描述
root	[IN] PlannerInfo *	查询优化模块的上下文信息结构体
sublink	[IN] SubLink *	要处理的子连接信息
available_rels	[OUT] Relids	输出参数，适用于子连接提升的表的集合
返回值	JoinExpr *	返回Semi Join类型的JoinExpr

对于 ANY 类型的子连接的提升，需要满足如下条件：

1）ANY 类型的子连接的左操作数如果不在 available_rels 中，子连接不能提升，具体可参考 pull_up_sublinks_jointree_recurse 函数对 JoinExpr 的处理，以及表 3-4 逻辑连接操作符和子连接提升的关系。

2）ANY 类型的子连接如果是“相关子连接”，即子连接中引用了父查询的列属性，子连接不能提升。

3）ANY 类型的子连接 SubLink->testexpr 中没有引用上一层的列，不能提升，因为这种情况下上层的表和子连接中的表不能构成连接关系。

```
postgres=# EXPLAIN SELECT * FROM STUDENT WHERE 1 > ANY(SELECT sno FROM SCORE);
```

版权所有，严禁传播，违者自负法律责任！


```
                              QUERY PLAN
------------------------------------------------------------------------------
 Result  (cost=22.85..43.85 rows=1100 width=46)
   One-Time Filter: (SubPlan 1)
   ->  Seq Scan on student  (cost=22.85..43.85 rows=1100 width=46)
   SubPlan 1
     ->  Materialize  (cost=0.00..40.60 rows=2040 width=4)
           ->  Seq Scan on score  (cost=0.00..30.40 rows=2040 width=4)
(6 rows)
```

4）ANY 类型的子连接中 SubLink->testexpr 中没有 Var，也就是没有列属性，子连接不能提升。

5）ANY 类型的子连接中 SubLink->testexpr 中如果含有易失性函数（PostgresQL 数据库中的函数有三种稳定性级别：VOLATILE、STABLE 和 IMMUTABLE，其中 VOLATILE 函数输入同样的参数会返回不同的结果，查询优化模块通常不对含有 VOLATILE 函数的表达式进行优化），子连接不能提升。

3.2.1.4　ANY 类型子连接的提升流程

ANY 类型的提升过程主要有 5 个步骤，我们以 SELECT sname FROM STUDENT WHERE sno > ANY (SELECT sno FROM SCORE)为例进行说明，该语句在子连接提升前的查询树（Query）如图 3-3 所示，子连接以 SubLink 的形式出现在查询树中。

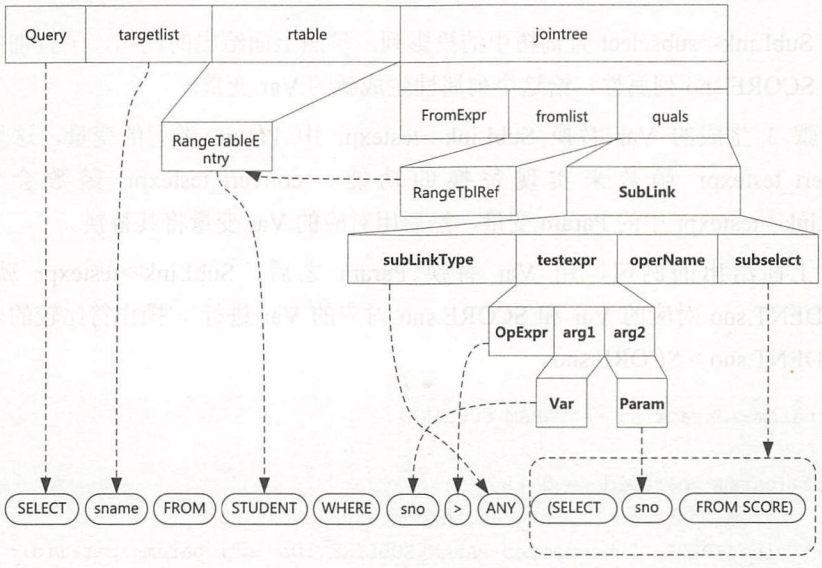


图 3-3　带有 Any 类型子连接的查询树内存结构图

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

1）SubLink->subselect 中存储的是子连接的子句对应的查询树，首先将这个查询树转变成 SubQuery 类型的 RangeTblEntry，参照上面给出的例句，这里的 SubLink->subselect 就是（SELECT sno FROM SCORE）生成的查询树，将这个查询树转换成子查询，加入上层查询树的 rtable 中。

```
//生成新的 RangeTblEntry 节点
rte = addRangeTableEntryForSubquery(pstate,
                                    subselect,
                                    makeAlias("ANY_subquery", NIL),
                                    false,
                                    false);
//加入到上层的 rtable 中，以获得下标值
parse->rtable = lappend(parse->rtable, rte);
```

2）SubLink->subselect 加入上层的 rtable 后，它就获得了新的 rtindex，根据这个 rtindex 生成 RangeTblRef。

```
parse->rtable = lappend(parse->rtable, rte);
//获得 rtindex
rtindex = list_length(parse->rtable);
//生成新的 RangeTableRef 节点
rtr = makeNode(RangeTblRef);
rtr->rtindex = rtindex;
```

3）提取 SubLink->subselect 查询树中的投影列，参照上面给出的例句，子查询中的投影列就是 SCORE.sno 列属性，给这个列属性生成新的 Var 变量。

4）用步骤 3 生成的 Var 替换 SubLink->testexpr 中 Param 类型的变量，这里通过调用 convert_testexpr 函数来实现替换的功能，convert_testexpr 函数会递归查找 SubLink->testexpr 中的 Param 变量，然后用对应的 Var 变量将其替换。

参考上面给出的例句，用 Var 替换 Param 之后，SubLink->testexpr 就转换成了 STUDENT.sno 对应的 Var 和 SCORE.sno 对应的 Var 进行'>'操作符比较的表达式，即 STUDENT.sno > SCORE.sno。

```
if (param->paramkind == PARAM_SUBLINK)
{
   if (param->paramid <= 0 ||
      param->paramid > list_length(context->subst_nodes))
      elog(ERROR, "unexpected PARAM_SUBLINK ID: %d", param->paramid);
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
    //Param 的编号和 Var 链表中的位置是一一对应的
    return (Node *) copyObject(list_nth(context->subst_nodes,
                                        param->paramid - 1));
}
```

5）这时即有了新的 RangeTblRef（步骤 2 生成），也有了新的 quals（步骤 4 生成），我们就可以生成新的 JoinExpr，新的 RangeTblRef 作为连接的 RHS 端，约束条件就是 SubLink->testexpr 转换后的表达式，注意这时候生成的 JoinExpr 是不完整的，它的 LHS 端目前是 NULL，LHS 的值由 pull_up_sublinks_qual_recurse 函数去填充。

```
result = makeNode(JoinExpr);
result->jointype = JOIN_SEMI; //连接类型是 Semi Join
result->isNatural = false;
result->larg = NULL;          //pull_up_sublinks_qual_recurse 函数去填充
result->rarg = (Node *) rtr; //SubLink.subselect 转换出来的子查询
result->usingClause = NIL;
result->quals = quals; //SubLink.testexpr 转换出来的表达式
result->alias = NULL;
result->rtindex = 0;
```

经过变换后的 SQL 语句我们可以写成（不能执行）：

```
SELECT sname FROM STUDENT SEMI JOIN (SELECT sno FROM SCORE) ANY_subquery WHERE STUDENT.sno > ANY_subquery.sno;
```

通过观察变换后的语句可以发现，ANY 类型的子连接在提升后出现在了 FROM 关键字之后，变成了一个范围表，也就是说变成了子查询。不过在随后查询优化器还会尝试提升子查询的，至于这种子查询是否会被提升，最终取决于它是不是满足子查询提升的条件。

3.2.1.5　EXISTS 类型子连接的提升条件

EXISTS 类型的子连接在 convert_EXISTS_sublink_to_join 函数中提升，该函数对查询树中的子连接进行了结构变换，将子连接变为和上层表的半连接，函数的参数说明如表 3-6 所示。

表 3-6　convert_EXISTS_sublink_to_join函数参数说明

参数名	参数类型	描述
root	[IN] PlannerInfo *	查询优化模块的上下文信息结构体
sublink	[IN] SubLink *	要处理的子连接信息
under_not	[IN] bool	用于区分EXIST和NOT EXISTS
available_rels	[OUT] Relids	输出参数，jtnode参数中涉及的表的集合
返回值	JoinExpr *	返回Semi Join或Anti Join类型的JoinExpr

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

EXISTS 类型的子连接和 ANY 类型的子连接不同，它没有左操作数，因此 EXISTS 类型的子连接能够提升的条件与 ANY 类型的子连接也不同。

1）EXISTS 类型的子连接的子句中如果包含通用表达式（CTE），子连接不能提升。

2）通过 simplify_EXISTS_query 函数来判断 EXISTS_SUBLINK 类型的子连接的子句是否“简单”，如果子句中包含集合操作、聚集操作、HAVING 子句等，子连接不能提升，否则对 SubLink 中的子句进行简化。

```
//判断 EXISTS_SUBLINK 子连接的子句是否“简单”，不满足其中的任意一项，子连接都不能提升
if (query->commandType != CMD_SELECT ||   //必须是 SELECT 子句
 query->setOperations ||                  //子句中不能带有集合操作
 query->hasAggs ||                        //子句中不能带有聚集函数
 query->groupingSets ||                   //不能有 group by grouping sets 这样的子句
 query->hasWindowFuncs ||                 //不能有窗口函数
 query->hasTargetSRFs ||                  //不能有 generate_series 这样的函数
 query->hasModifyingCTE ||                //不能有带有 UPDATE、INSERT 语句的通用表达式
 query->havingQual ||                     //不能有 having 子句
 query->limitOffset ||                    //不能带有 limit 子句
 query->rowMarks)                         //不能是 SELECT …for update 子句
     return false;

//如果通过了上面的检查，那么还可以尝试简化这个子连接
//由于 EXISTS 类型的子连接具有找到一个即可的特点
//因此 LIMIT 子句如果只是对结果进行限制，这个子句是可以消除的
if (query->limitCount)
{……}

//下面这些也不会影响 EXISTS 类型的子连接的结果，也可以简化掉
query->targetList = NIL;
query->groupClause = NIL;
query->windowClause = NIL;
query->distinctClause = NIL;
query->sortClause = NIL;
query->hasDistinctOn = false;
```

3）EXISTS 类型的子连接的子句中，如果约束条件（WHERE/ON）中没有包含上层的列属性（Var），这种情况无须将子连接提升为 JOIN，通常直接对子连接单独生成子计划求解代价更低。例如有 SQL 语句 SELECT * FROM STUDENT WHERE EXISTS (SELECT * FROM SCORE)，这个子连接中没有引用父查询的列属性，因此无须提升。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

4）EXISTS 类型的子连接的子句中如果除了约束条件之外的其他表达式如果引用了上层父查询的列属性（Var），子连接不能提升。同时，子连接的约束条件中则必须包含上层父查询的列属性，否则不能提升。例如下面的示例，子连接的约束条件中没有包含父查询的列属性，从执行计划可以看出子连接没有提升。

```
postgres=# EXPLAIN SELECT * FROM STUDENT WHERE EXISTS (SELECT sno FROM SCORE WHERE
sno = 1);
                        QUERY PLAN
------------------------------------------------------------
 Result  (cost=1.01..2.08 rows=7 width=19)
   One-Time Filter: $0
   InitPlan 1 (returns $0)
     ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=0)
           Filter: (sno = 1)
   ->  Seq Scan on student  (cost=1.01..2.08 rows=7 width=19)
(6 rows)
```

5）EXISTS 类型的子连接的约束条件（Whereclause）中如果含有易失性函数，子连接不能提升。

3.2.1.6　EXISTS 类型子连接的提升流程

我们以 SQL 语句 SELECT sno FROM STUDENT WHERE EXISTS (SELECT sno FROM SCORE WHERE sno > STUDENT.sno)为例来说明 EXISTS 类型的子连接的提升过程，这个查询语句形成的查询树的结构如图 3-4 所示。

在 EXISTS 类型的子连接提升的过程中，最主要的是对 SubLink->subselect 的处理，一方面 SubLink->subselect->rtable 将被提升到上层，和上层表形成 SemiJoin 或 AntiJoin 关系，另一方面 SubLink->subselect->FromExpr->quals 也会被提升到上层变成约束条件，在将子连接的范围表和约束条件提升后，还需要调整 SubLink->subselect 查询树中的 Var 变量中的 varno 和 varlevelsup，下面是提升的具体流程。

1）将 SubLink->subselect 子查询中的 FromExpr->quals 独立保存起来，形成 whereClause，这部分将来要做约束条件，然后将 FromExpr->quals 设置为 NULL。

```
whereClause = subselect->jointree->quals;
subselect->jointree->quals = NULL;
```

2）对 SubLink->subselect 和 whereClause 分别处理，目前它们中的 Var 变量的 varno 是根据 SubLink->subselect->rtable 确定的，如果这些 Var 被提升到上层，它们的 varno 就要做出调整，由于要将子连接中的范围表追加到上层父查询的范围表的链表中（rtable 链表），

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

因此子连接中的范围表的 rtindex 需要增加上层父查询的范围表链表的长度，同时对 SubLink->subselect 中的 RangeTableRef 的 rtindex 也要按照新的 rtindex 做调整。

3）对 Sublink->subselect 和 whereClause 分别处理，如果引用了上层表的列属性，那么这个 Var 的 varlevelup 的值是 1，表示它是上一层的某个表的列属性，如果子连接被提升，这个 Var 的 varlevelsup 应该做出调整，修改成 0，表 3-7 是子连接提升前后 Var 中的值的对比表。

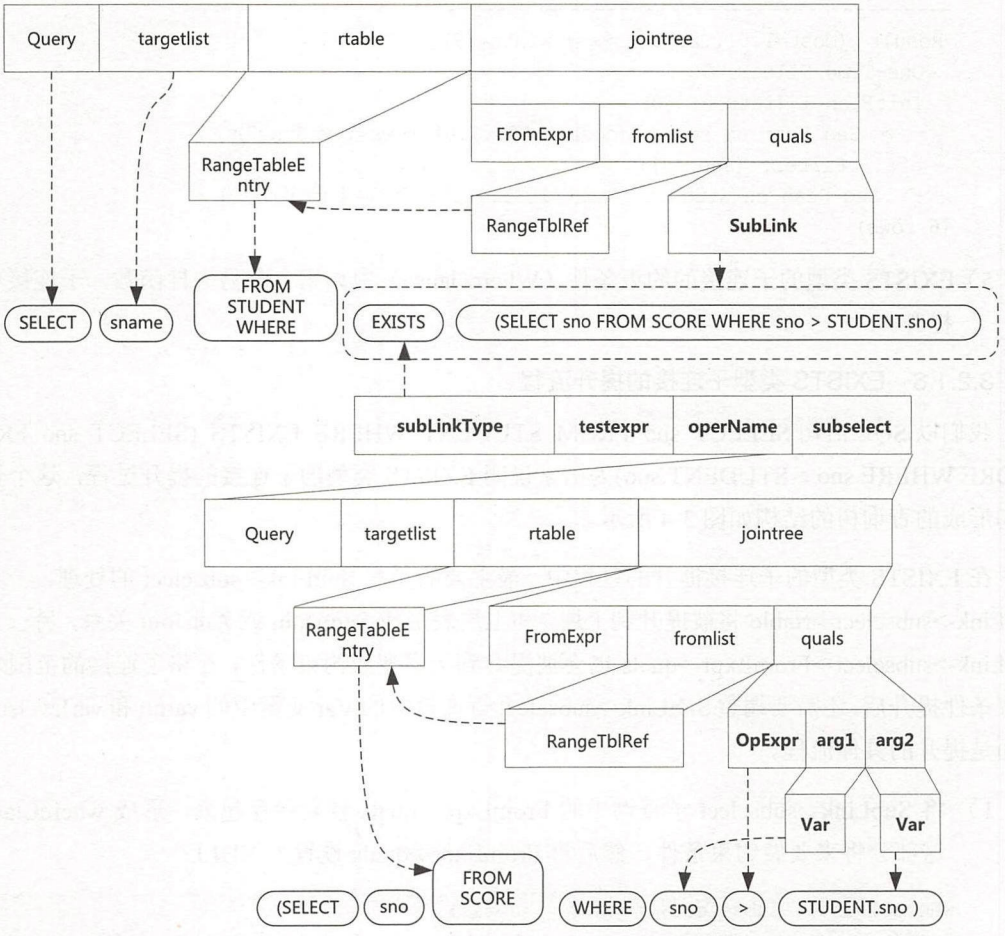


图 3-4　带有 EXISTS 类型的子连接的查询树的内存结构图

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

表 3-7　子连接提升前后Var结构对比表

<table>
<tr><td colspan="4">示例查询语句：SELECT sno FROM STUDENT WHERE EXISTS (SELECT sno FROM SCORE WHERE sno > STUDENT.sno)
在源代码的whereClause变量中保存的是子连接中的约束条件，也就是WHERE sno > STUDENT.sno，在子查询树的quals中，第一个Var是SCORE.sno，它是属于子连接的列属性，因此调整了varno（即rtindex），第二个Var变量是STUDENT.sno，它是上层父查询的列属性但是处在子连接中，因此未提升之前它的varlevelsup的值是1，而提升之后就变成了同层引用，因此就变成了0</td></tr>
<tr><td colspan="2">SCORE.sno</td><td colspan="2">STUDENT.sno</td></tr>
<tr><td>{VAR
:varno 1
:varattno 1
:vartype 23
:vartypmod -1
:varcollid 0
:varlevelsup 0
:varnoold 1
:varoattno 1
:location 76
}</td><td>{VAR
:varno 2
:varattno 1
:vartype 23
:vartypmod -1
:varcollid 0
:varlevelsup 0
:varnoold 1
:varoattno 1
:location 76
}</td><td>{VAR
:varno 1
:varattno 1
:vartype 23
:vartypmod -1
:varcollid 0
:varlevelsup 1
:varnoold 1
:varoattno 1
:location 82
}</td><td>{VAR
:varno 1
:varattno 1
:vartype 23
:vartypmod -1
:varcollid 0
:varlevelsup 0
:varnoold 1
:varoattno 1
:location 82
}</td></tr>
</table>

4）将子连接的 rtable 附加到上层父查询的 rtable。

5）创建新的 jointree，用 SubLink->subselect 中的 jointree 作为右参数（rarg），用 whereClause 作为新的 quals。

参考示例语句，其子连接提升之后的语句形式为（不能执行）：

```
SELECT sno FROM STUDENT SEMI JOIN SCORE WHERE SCORE.sno > STUDENT.sno;
```

不同于 ANY 类型的子连接，EXISTS 类型的子连接会彻底提升为表与表直接连接的方式，而不是先转换成子查询。

3.2.2　提升子查询

子查询和子连接不同，它出现在 RangeTableEntry 中，它存储的是一个子查询树，如果这个子查询树不被提升，则经过查询优化之后形成一个子执行计划，上层执行计划和子查询计划做嵌套循环得到最终结果，在这个过程中，查询优化模块对这个子查询所能做的优化选择较少。如果这个子查询被提升，转换成与上层的连接（Join），由于查询优化模块对连接操作的优化

非卖品，仅供非商业用途或交流学习使用

做了很多工作，因此可能获得更好的执行计划。

如图 3-5 所示，子查询的提升主要在 pull_up_subqueries 函数中完成，该函数又调用了递归遍历函数 pull_up_subqueries_recurse 函数，pull_up_subqueries_recurse 函数则对 Query->jointree 进行递归遍历，针对 Query->jointree 中可能出现的 RangeTblRef、FromExpr、JoinExpr 进行了不同的处理。

- RangeTblRef 结构体又细分成 RTE_SUBQUERY(simple)、RTE_SUBQUERY(union)、RTE_VALUES，对它们需要分别做不同的处理，RTE_SUBQUERY(simple)调用 pull_up_simple_subquery 函数进行处理，RTE_SUBQUERY(union)调用 pull_up_simple_union_all 函数进行处理，RTE_VALUES 调用 pull_up_simple_values 进行处理。
- 对于 FromExpr，因为 FromExpr->fromlist 中的范围表默认是内连接，所以只需要遍历 FromExpr->fromlist 中的节点，这里主要判断下层的节点是否存在可删除的情况。
- 对于 JoinExpr，因为有外连接和半连接的出现，有些子查询的提升会受到限制，例如在 PostgreSQL 8.4 之前，如果子查询处于外连接 Nullable-side，且投影列中有不严格的函数，那么就无法提升，目前这类子查询通过借用 PlaceHolderVar 结构体实现了提升（注：一个函数如果输入参数是 NULL，它的执行结果也是 NULL，那么这个函数就是严格的，具体可以参考第 3 章中外连接消除的章节，另外，PlaceHolderVar 的说明可以参考第 4 章中的说明）。

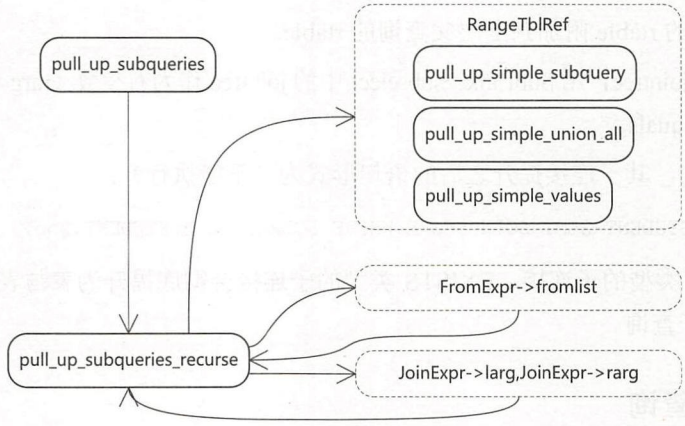


图 3-5　子查询提升函数调用关系

pull_up_subqueries_recurse 函数是子查询提升的“主力”函数，它的参数说明如表 3-8 所示。

非卖品，仅供非商业用途或交流学习使用

表 3-8　子查询提升参数说明

参数名	参数类型	描述
root	[IN] PlannerInfo *	查询优化的上下文信息，贯穿整个查询优化模块
jtnode	[IN]Node*	需要递归处理的Node，可能是FromExpr、JoinExpr或者RangeTblRef
lowest_outer_join	[IN] JoinExpr *	如果在JoinExpr中存在外连接，则它是整个外连接的Node节点，包含内表和外表
lowest_nulling_outer_join	[IN] JoinExpr *	如果在JoinExpr中存在外连接，则它是外连接中的nullable side，如果是左外连接，则这里是右表，如果是右外连接，这里是左表，如果是全外连接，这里是整个外连接的Node节点
containing_appendrel	[IN] AppendRelInfo *	对于RangeTblRef中存在RTE_SUBQUERY(union)的情况，调用pull_up_simple_union_all函数，分别处理union操作下的子查询
deletion_ok	[IN] bool	和 root->hasDeletedRTE 结合使用，对于RTE_VALUES类型的子查询和无范围表的情况，子查询提升后会产生可以删除的节点

3.2.2.1　RTE_SUBQUERY(SIMPLE)

对于 RTE_SUBQUERY（simple）类型的子查询，我们先看一下它的提升条件，首先要求子查询的类型是“简单（simple）”的，所谓的简单，需要满足如下条件：

a）做一个确认，必须是一个子查询树，而且是 SELECT 查询语句。

b）子查询树的 subQuery->setOperations 必须是 NULL，如果不是 NULL，应该先交给 pull_up_simple_union_all 函数去处理。

c）不能包含聚集操作、窗口函数、GROUP 操作等，在子连接提升的时候我们已经见过类似的条件。

```
if (subquery->hasAggs ||
    subquery->hasWindowFuncs ||
    subquery->hasTargetSRFs ||
    subquery->groupClause ||
    subquery->groupingSets ||
    subquery->havingQual ||
    subquery->sortClause ||
    subquery->distinctClause ||
    subquery->limitOffset ||
    subquery->limitCount ||
```


非卖品，仅供非商业用途或交流学习使用

```
    subquery->hasForUpdate ||
    subquery->cteList)
        return false;
```

d）如果没有范围表，那么在无约束条件或者满足删除条件或者不是外连接的情况下才能提升，这是因为在物理优化阶段所有的表都会建立一个 RelOptInfo，如果空范围表的子查询不提升，那么可以用 RelOptInfo 来表示子查询，然后可以将这个 RelOptInfo 优化成一个 Result 计划节点，如果将空范围表提升上来，那么无法用 RelOptInfo 表示它，请看下面的示例：

```
-- 有约束条件的例子，假如提升上来，不知道该如何安置这个约束条件
postgres=# EXPLAIN SELECT * FROM STUDENT, LATERAL (SELECT 1 AS a WHERE STUDENT.sno >
1) as q;
                        QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=0.00..480.00 rows=10000 width=15)
   ->  Seq Scan on student  (cost=0.00..155.00 rows=10000 width=11)
   ->  Result  (cost=0.00..0.01 rows=1 width=4)
         One-Time Filter: (student.sno > 1)
(4 rows)

-- 不是 deletion_ok 的情况，同时也是外连接的情况
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN (SELECT 1 AS a) as q ON TRUE;
                        QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..280.02 rows=10000 width=15)
   ->  Seq Scan on student  (cost=0.00..155.00 rows=10000 width=11)
   ->  Materialize  (cost=0.00..0.03 rows=1 width=4)
         ->  Result  (cost=0.00..0.01 rows=1 width=4)
(4 rows)
```

e）Lateral 语义支持在子查询中引用上一层的表，但是如果引用的是更上层的表，可能会出现问题。例如下面的语句中，如果子查询提升，TEST_A.a > 1 这个约束条件的提升可能会改变查询结果（如果子查询没有提升，在物理优化阶段会生成 SubqueryScan 路径，注意：虽然下面示例的执行计划中没有 SubqueryScan 这个执行路径，但是实际上 SubqueryScan 这个物理路径是生成了的，只不过在查询优化的最后阶段——清理执行计划时又优化掉了，执行计划清理见第 10 章）。

```
postgres=# EXPLAIN SELECT * FROM TEST_A LEFT JOIN (TEST_B LEFT JOIN LATERAL (SELECT
* FROM TEST_C WHERE TEST_A.a > 1) AS c ON TRUE) ON TRUE;
```


非卖品，仅供非商业用途或交流学习使用

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..809535.32 rows=5120000 width=48)
   ->  Seq Scan on test_a  (cost=0.00..57.56 rows=256 width=16)
   ->  Nested Loop Left Join  (cost=0.00..2962.02 rows=20000 width=32)
         ->  Seq Scan on test_b  (cost=0.00..54.02 rows=2 width=16)
         ->  Materialize  (cost=0.00..2683.00 rows=10000 width=16)
               ->  Result  (cost=0.00..2533.00 rows=10000 width=16)
                     One-Time Filter: (test_a.a > 1)
                     ->  Seq Scan on test_c  (cost=0.00..2533.00 rows=10000 width=16)
(8 rows)
```

另外，如果子查询的投影中包含了上层外连接的列属性，子查询也不能提升，例如下面的 SQL 语句中子查询就没有提升（没有 SubqueryScan 的原因同上）。

```
postgres=# EXPLAIN SELECT * FROM TEST_A LEFT JOIN (TEST_B LEFT JOIN LATERAL (SELECT
*,TEST_A.a FROM TEST_C) AS c ON TRUE) ON TRUE;
                              QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..809534.68 rows=5120000 width=52)
   ->  Seq Scan on test_a  (cost=0.00..57.56 rows=256 width=16)
   ->  Nested Loop Left Join  (cost=0.00..2962.02 rows=20000 width=36)
         ->  Seq Scan on test_b  (cost=0.00..54.02 rows=2 width=16)
         ->  Materialize  (cost=0.00..2683.00 rows=10000 width=20)
               ->  Seq Scan on test_c  (cost=0.00..2533.00 rows=10000 width=20)
(6 rows)
```

f）如果有易失性函数也不能提升。

在满足“简单”的同时，如果是从 pull_up_simple_union_all 函数递归过来的子查询（containing_appendrel ！= NULL），那么需要满足没有约束条件、只能有一个范围表（通过 is_safe_append_member 函数判断）才能提升。

假如有这样一个 SQL 语句：

```
SELECT * FROM STUDENT, (SELECT * FROM SCORE WHERE sno > 10) sc WHERE STUDENT.SNO = sc.sno;
```

我们可以直观地判断这是一个 RTE_SUBQUERY(simple) 类型的子查询，按照 RTE_SUBQUERY(simple)类型的子查询提升的条件来逐一对比：a) 子查询本身是一个 SELECT 语句；b) 没有 subquery->setOperations，也就是说不带有集合操作；c) 没有聚集函数等；d) 有一个范围表 SCORE；e) 不存在 LATERAL 语义；f) 也没有易失性函数，同时 containing_appendrel 也一定是 NULL。综合上述条件这个子查询是能够提升的，而且通过 pull_up_simple_subquery

函数来直接提升，下面是这个 SQL 语句的执行计划。

```
postgres=# EXPLAIN VERBOSE SELECT * FROM STUDENT, (SELECT * FROM SCORE WHERE sno
 > 10) sc WHERE STUDENT.SNO = sc.sno;
                              QUERY PLAN
-----------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..107.75 rows=24 width=23)
   Output: student.sno, student.sname, student.ssex, score.sno, score.cno, score.degree
   Inner Unique: true
   Join Filter: (student.sno = score.sno)
   -> Seq Scan on public.score  (cost=0.00..35.50 rows=680 width=12)
        Output: score.sno, score.cno, score.degree
        Filter: (score.sno > 10)
   -> Materialize  (cost=0.00..1.11 rows=7 width=11)
        Output: student.sno, student.sname, student.ssex
        -> Seq Scan on public.student  (cost=0.00..1.07 rows=7 width=11)
             Output: student.sno, student.sname, student.ssex
(11 rows)
```

通过观察执行计划可以看出，子查询中的表 SCORE 提升上来和上层的表 STUDENT 做了 NestedLoop Join，根据上面的执行计划可以推算等价的 SQL 语句。

```
postgres=# EXPLAIN VERBOSE SELECT * FROM STUDENT,SCORE WHERE SCORE.sno > 10 AND
STUDENT.SNO = SCORE.sno;
                              QUERY PLAN
-----------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..107.75 rows=24 width=23)
   Output: student.sno, student.sname, student.ssex, score.sno, score.cno, score.degree
   Inner Unique: true
   Join Filter: (student.sno = score.sno)
   -> Seq Scan on public.score  (cost=0.00..35.50 rows=680 width=12)
        Output: score.sno, score.cno, score.degree
        Filter: (score.sno > 10)
   -> Materialize  (cost=0.00..1.11 rows=7 width=11)
        Output: student.sno, student.sname, student.ssex
        -> Seq Scan on public.student  (cost=0.00..1.07 rows=7 width=11)
             Output: student.sno, student.sname, student.ssex
(11 rows)
```

根据执行计划我们可以看出子查询的投影列（targetlist）和范围表（rtable）都做了提升，投影列提升到上层做投影，范围表则提升到上层直接做连接（Join）操作。下面再看一个不能提升的例子，假如有如下语句：

```
SELECT * FROM STUDENT st LEFT JOIN (SELECT sno, COALESCE(degree,60) FROM SCORE) sc ON
st.sno=sc.sno
```

它的执行情况如下：

```
postgres=# SELECT sno, sname, ssex FROM STUDENT;
 sno |  sname   | ssex
-----+----------+------
   1 | zhangsan |    1
   2 | lisi     |    1
(2 rows)

postgres=# SELECT sno, cno, degree FROM SCORE;
 sno | cno | degree
-----+-----+--------
   1 |   1 |     36
(1 row)

postgres=# SELECT * FROM STUDENT st LEFT JOIN (SELECT sno, COALESCE(degree,60) FROM SCORE)
sc ON st.sno=sc.sno;
 sno |  sname   | ssex | sno | coalesce
-----+----------+------+-----+----------
   1 | zhangsan |    1 |   1 |       36
   2 | lisi     |    1 |     |
(2 rows)
```

用反证法，假设这个子查询是能提升的，对其进行强制提升，得到的查询结果如下：

```
-- 强制提升
postgres=# SELECT st.sno, sname, ssex, sc.sno, COALESCE(degree,60) FROM STUDENT st LEFT JOIN
SCORE sc ON st.sno=sc.sno;
 sno |  sname   | ssex | sno | coalesce
-----+----------+------+-----+----------
   1 | zhangsan |    1 |   1 |       36
   2 | lisi     |    1 |     |       60
(2 rows)
```

COALESCE 函数对 NULL 值做了处理，导致外连接补的 NULL 值也被处理了，所以强制提升之后两个查询就不再等价。

PostgreSQL 数据库在 8.4 之前的版本对这一类子查询是不能提升的，在 8.4 版本引入了 PlaceHolderVar 结构体，用来对这种情况进行处理。PlaceHolderVar 结构体只在查询优化阶段使用，它在逻辑优化的部分生成，在生成执行计划（Plan）的时候会被消除掉，它的主要作用是

在查询优化的阶段标识出像 coalesce 这种特殊的表达式，查询优化模块会单独为 PlaceHolderVar 做特殊的处理，下面看一下示例语句的执行计划。

```
-- 需要注意这里虽然有两个 COALESCE(score.degree, 60)，但是实际上是上层
-- 的 COALESCE(score.degree, 60)对下层的引用，
-- 上层的 COALESCE(score.degree, 60)实际上是一个 Var 结构体，
-- EXPLAIN 语句对这种 Var 进行了处理，所以上层也显示成 COALESCE(score.degree, 60))
postgres=# EXPLAIN VERBOSE SELECT * FROM STUDENT st LEFT JOIN (SELECT sno, COALESCE(degree,60)
FROM SCORE) sc ON st.sno=sc.sno;
                            QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..250.77 rows=71 width=19)
   Output: st.sno, st.sname, st.ssex, score.sno, (COALESCE(score.degree, 60))
   Join Filter: (st.sno = score.sno)
   -> Seq Scan on public.student st  (cost=0.00..1.07 rows=7 width=11)
        Output: st.sno, st.sname, st.ssex
   -> Materialize  (cost=0.00..40.60 rows=2040 width=8)
        Output: score.sno, (COALESCE(score.degree, 60))
        -> Seq Scan on public.score  (cost=0.00..30.40 rows=2040 width=8)
             Output: score.sno, COALESCE(score.degree, 60)
(9 rows)

-- 可以看到在对 SCORE 表进行扫描的时候，直接把投影列传递给了连接操作，
-- COALESCE(sc.degree, 60)在连接操作之后进行计算
postgres=# EXPLAIN VERBOSE SELECT st.sno, sname, ssex, sc.sno, COALESCE(degree,6
0) FROM STUDENT st LEFT JOIN SCORE sc ON st.sno=sc.sno;
                            QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..250.77 rows=71 width=19)
   Output: st.sno, st.sname, st.ssex, sc.sno, COALESCE(sc.degree, 60)
   Join Filter: (st.sno = sc.sno)
   -> Seq Scan on public.student st  (cost=0.00..1.07 rows=7 width=11)
        Output: st.sno, st.sname, st.ssex
   -> Materialize  (cost=0.00..40.60 rows=2040 width=8)
        Output: sc.sno, sc.degree
        -> Seq Scan on public.score sc  (cost=0.00..30.40 rows=2040 width=8)
             Output: sc.sno, sc.degree
(9 rows)
```

我们就以 SELECT * FROM STUDENT st LEFT JOIN (SELECT sno, COALESCE(degree,60) FROM SCORE) sc ON st.sno=sc.sno 为例来看一下 RTE_SUBQUERY(simple)的提升流程。

1）子查询中的 Var 变量的 varno 需要调整，因为子查询中 Var->varno 都是基于子查询树中 subquery->rtable 链表的 rtindex，当子查询树中的表提升到父查询之后，varno 就需要进行相应的调整。

如果子查询中引用了父查询表的某列，那么代表这个列的 Var 中的 Var->varlevelsup 的值应该是 1，也就是说这个 Var 是“上 1 层”的父查询中的列属性，在子查询提升之后，查询树就没有了上下层的关系，因此这个 Var->varlevelsup 需要被调整为 0。

```
//调整 varno
rtoffset = list_length(parse->rtable);
OffsetVarNodes((Node *) subquery, rtoffset, 0);
OffsetVarNodes((Node *) subroot->append_rel_list, rtoffset, 0);

//调整 varlevelsup
IncrementVarSublevelsUp((Node *) subquery, -1, 1);
IncrementVarSublevelsUp((Node *) subroot->append_rel_list, -1, 1);
```

2）这时子查询中的 Var 已经修正好了，如果父查询中有引用子查询的列，那么直接用子查询中修正好的 Var 替换父查询中的 Var 即可，替换使用的是 pullup_replace_vars 函数，这个函数的主要参数是 pullup_replace_vars_context 结构体。

```
typedef struct pullup_replace_vars_context
{
   PlannerInfo *root;  //查询优化模块的上下文结构体
   List     *targetlist;  //子查询中的投影列，用这里的 Var 替换上层的 Var
   RangeTblEntry *target_rte; //子查询的 RangeTblEntry 结构体

   //子查询 subquery->jointree 中涉及的表的 relids 集合
   //主要用于在包含 LATREAL 语义的时候，
   //如果子查询的投影列中引用了上层表的列，那么就需要使用 PlaceHolderVar
   Relids relids;
   bool *outer_hasSubLinks;  //外层查询有子连接
   int     varno; //当前子查询的 RangeTblEntry 对应的 rtindex

   //是否需要使用 PlaceHolderVar 结构体
   bool    need_phvs;
   bool    wrap_non_vars; // 它的作用是指明下层查询中只要出现了表达式，
                  //就使用 PlaceHolderVar 进行封装，目前主要是
                  //针对 Appendrel 子表的下的表达式

   //给每个 Var 生成结构体之后会记录到这里，防止重复生成
```

```
    Node    **rv_cache;
} pullup_replace_vars_context;
```

在有了 LATERAL 语义之后，子查询的投影列也能引用外层的表中的列，这也需要替换成 PlaceHolderVar，例如： SELECT * FROM STUDENT st LEFT JOIN (SELECT sno, **st.sname** FROM SCORE) sc on st.sno=sc.sno;子查询中的 st.sname 就会被替换成 PlaceHolderVar。

3）将子查询中的 subquery->rtable 附加到父查询的 Query->rtable 链表中，同时，subquery->rowMarks 也会被提升到上层。

```
parse->rtable = list_concat(parse->rtable, subquery->rtable);
parse->rowMarks = list_concat(parse->rowMarks, subquery->rowMarks);
```

3.2.2.2 RTE_SUBQUERY(UNION)

对于 RTE_SUBQUERY(union)类型的子查询，它的主要问题是在子查询中如果有 UNION ALL 操作，就会出现“子子查询”，例如下面的 SQL 语句：

```
SELECT * FROM SCORE sc, (SELECT * FROM STUDENT WHERE sno =1 UNION ALL SELECT * FROM STUDENT WHERE sno=2) st WHERE st.sno > 0;
```

这个 SQL 语句查询树中有三个层次，层次关系如图 3-6 所示。

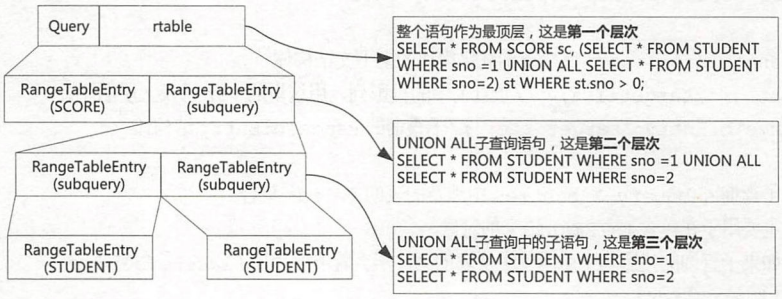


图 3-6 带有 UNION 类型的子查询的查询树结构图

作为 UNION ALL 联合查询，它的效果和 PostgreSQL 数据库的继承表的效果是相似的，UNION ALL 联合查询是对两边的语句的结果集进行整合，继承表则是整合子表的查询结果。

```
CREATE TABLE INH_P(a INT, b TEXT);
CREATE TABLE INH_C1() INHERITS(INH_P);
CREATE TABLE INH_C2() INHERITS(INH_P);

postgres=# EXPLAIN SELECT * FROM INH_P;
```

```
                        QUERY PLAN
------------------------------------------------------------------
 Append  (cost=0.00..45.40 rows=2541 width=36)
   ->  Seq Scan on inh_p  (cost=0.00..0.00 rows=1 width=36)
   ->  Seq Scan on inh_c1  (cost=0.00..22.70 rows=1270 width=36)
   ->  Seq Scan on inh_c2  (cost=0.00..22.70 rows=1270 width=36)
(4 rows)
```

从示例的执行计划可以看出，在查询父表 INH_P 的时候，子表以 Append 子节点的方式也进行了查询。继承表的子表在 expand_inherited_tables 函数中进行展开，然后生成新的 AppendRelInfo 结构体。

```
typedef struct AppendRelInfo
{
    NodeTag  type;
    Index         parent_relid;  //父表的 rtindex
    Index         child_relid;     //子表的 rtindex

    Oid       parent_reltype; //创建父表的时候在 pg_type 中生成的行类型
    Oid       child_reltype;  //创建子表的时候在 pg_type 中生成的行类型

    List    *translated_vars;   /* Expressions in the child's Vars */
    Oid       parent_reloid;    /* OID of parent relation */
} AppendRelInfo;
```

我们提升的主要目的是将三个层次变成两个层次，第三个层次的“子子查询”以 Append Relation 的形式提升一层，如下面的示例所示：

```
postgres=# EXPLAIN SELECT * FROM SCORE sc, (SELECT * FROM STUDENT WHERE sno =1 UNION ALL SELECT
* FROM STUDENT WHERE sno=2) st WHERE st.sno > 0;
                              QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..83.64 rows=4080 width=23)
   ->  Seq Scan on score sc  (cost=0.00..30.40 rows=2040 width=12)
   ->  Materialize  (cost=0.00..2.24 rows=2 width=11)
       -> Append  (cost=0.00..2.23 rows=2 width=11)
          ->Seq Scan on student  (cost=0.00..1.11 rows=1 width=11)
                Filter: ((sno > 0) AND (sno = 1))
          ->Seq Scan on student student_1(cost=0.00..1.11 rows=1 width=11)
                Filter: ((sno > 0) AND (sno = 2))
(8 rows)
```

子查询的提升需要满足一些条件才能进行，这些条件的判断在 is_simple_union_all 函数中实现，分别如下：

a）必须是 SELECT 查询语句。

b）subquery->setOperations 必须有值，且集合操作的操作符必须是 UNION ALL。

c）子查询中不能有 ORDER BY、LIMIT/OFFSET 等。

```
/* Can't handle ORDER BY, LIMIT/OFFSET, locking, or WITH */
if (subquery->sortClause ||
    subquery->limitOffset ||
    subquery->limitCount ||
    subquery->rowMarks ||
    subquery->cteList)
        return false;
```

d）SetOperationStmt 中的子语句必须包含相同类型的投影列（is_simple_union_all_recurse 函数）。

RTE_SUBQUERY(union)类型的子查询会调用 pull_up_union_leaf_queries 再做递归遍历，找到其中的 RangeTblEntry 进行处理，调用关系如图 3-7 所示。

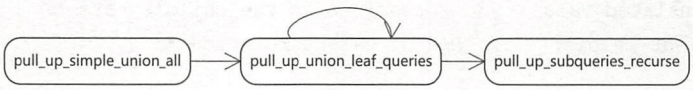


图 3-7　UNION 类型的子查询提升的函数调用关系

我们参照一个例子来看一下提升的具体流程。

```
SELECT *
FROM SCORE SC,
    LATERAL(SELECT * FROM STUDENT WHERE SNO = 1
          UNION ALL
          SELECT * FROM STUDENT WHERE SNO = SC.SNO) ST
WHERE ST.SNO > 0;
```

1）提升的主要目的是把三个层次变成两个层次，那么如果“子子查询”中引用了顶层的列属性，那么这些变量应该提升一个层次（IncrementVarSublevelsUp_rtable(rtable, -1, 1);），对应上面的示例语句，sc.sno 就引用了第一个层次中的列表量，它的 Var->varlevlesup 的原值是 2，子查询提升之后应该变成 1。

2）子查询提升之后，将原来在第二个层次的 LATERAL 语义下发给每个第三层次的子句，

参照上面的示例语句，也就是（SELECT * FROM STUDENT WHERE SNO = 1）和（SELECT * FROM STUDENT WHERE SNO = SC.SNO）这两个子查询都变成 LATERAL 的。

3）把第三层次的两个 RangeTblEntry——也就是（SELECT * FROM STUDENT WHERE SNO = 1）和（SELECT * FROM STUDENT WHERE SNO = SC.SNO）两个子查询附加到第一层的 Query->rtable 链表中。

```
/*
 * Append child RTEs to parent rtable.
 */
root->parse->rtable = list_concat(root->parse->rtable, rtable);
```

4）开始对 subquery->setOperations 进行遍历（pull_up_union_leaf_queries 函数），为其中的每个子查询生成一个 AppendRelInfo 节点，参照上面的示例语句，也就是为（SELECT * FROM STUDENT WHERE SNO = 1）和（SELECT * FROM STUDENT WHERE SNO = SC.SNO）生成两个 AppendRelInfo 节点。

```
//这里的 RangeTblRef 中的 rtindex 是之前位置的 rtindex
//在第 3 步骤中已经将 RangeTblEntry 附加到顶层，所以需要调整 rtindex
RangeTblRef *rtr = (RangeTblRef *) setOp;
int            childRTindex;
AppendRelInfo *appinfo;

//调整附加到顶层之后的 rtindex
childRTindex = childRToffset + rtr->rtindex;

//创建 AppendRelInfo 节点
appinfo = makeNode(AppendRelInfo);
appinfo->parent_relid = parentRTindex;
appinfo->child_relid = childRTindex;
appinfo->parent_reltype = InvalidOid;
appinfo->child_reltype = InvalidOid;

//第二个层次中的 Query->targetlist 中的 varno 需要调整位置
make_setop_translation_list(setOpQuery, childRTindex,
                            &appinfo->translated_vars);
appinfo->parent_reloid = InvalidOid;
root->append_rel_list = lappend(root->append_rel_list, appinfo);

//生成新的 RangeTblRef
```

```
rtr = makeNode(RangeTblRef);
rtr->rtindex = childRTindex;

//递归处理，提升子查询
 (void) pull_up_subqueries_recurse(root, (Node *) rtr,
                                   NULL, NULL, appinfo, false);
```

3.2.2.3 RTE_VALUES

RTE_VALUES 类型的子查询的 RangeTblEntry 中不止有 RTE_VALUES，而且包含 RTE_SUBQUERY，它的层次关系是：RTE_SUBQUERY 封装 RTE_VALUES，也就是说 RTE_VALUES 是保存在 RTE_SUBQUERY 的查询树 subquery->rtable 中的，因此它的提升的过程中可能经历两次提升，一次是子查询 RTE_SUBQUERY(SIMPLE)的提升，另一次是 RTE_VALUES 的提升。下面看一个 SQL 语法中带有 VALUES 子查询的例子。

```
//带有 VALUES 子查询，子查询的值直接体现在 SQL 语句中
SELECT sname, sc.sno, sc.cno FROM STUDENT, (VALUES(1,2)) AS sc(sno, cno) WHERE sc.sno =
STUDENT.sno;

// "(VALUES(1,2)) AS sc(sno, cno)"这部分子句在查询树中对应的内容
//(注:省略了一些不重要的变量)
{RTE  //RangeTblEntry 结构体
:alias
  {ALIAS
  :aliasname sc          //语句中的 sc(sno, cno)
  :colnames ("sno" "cno")  //语句中的 sc(sno, cno)
  }
:rtekind 1  //RTE_SUBQUERY == 1，代表这是一个子查询，这个子查询封装了 RTE_VALUES
:subquery   //这是 VALUES 对应的子查询
  {QUERY       //VALUES 对应的查询树
  :commandType 1  //CMD_SELECT == 1
  :rtable ( //这时候开始出现范围表的值，是一个 RangeTblEntry，是 RTE_VALUES 类型
    {RTE
    :alias <>
    :eref
      {ALIAS
      :aliasname *VALUES*
      :colnames ("column1" "column2") //这里没有给 VALUES 类型命名，在上面有
      }
    :rtekind 5 //RTE_VALUES == 5
    :values_lists (( //values_lists 里存储的是(VALUES(1,2))
```

```
        {CONST
        :constvalue 4 [ 1 0 0 0 ] //(VALUES(1,2))
        }
        {CONST
        :constvalue 4 [ 2 0 0 0 ] //(VALUES(1,2))
        }
    ))
    :coltypes (o 23 23) //VALUES 两列都是 int 类型（23 代表 int 类型，见 pg_type.h）
    }
  )
  :jointree
    {FROMEXPR
    :fromlist (
      {RANGETBLREF
      :rtindex 1 //引用 RTE_VALUES 类型的 RangeTblEntry
      }
    )
    :quals <>
  } //VALUES 对应的子查询结束
}//封装 VALUES 的子查询结束
```

由于 RTE_VALUES 是由 RTE_SUBQUERY 封装的，因此 RTE_VALUES 的提升可以包含两个步骤，第一个步骤是 RTE_VALUES 的提升，转换成等价的语句就是：

```
SELECT sname, sc.sno, sc.cno FROM STUDENT, (SELECT 1,2) AS sc(sno, cno) WHERE sc.sno =
STUDENT.sno;
```

这样 RTE_VALUES 就提升到了 RTE_SUBQUERY 的 targetList 里，RTE_VALUES 就被消除掉了，从原来的 3 个层次变成了 2 个层次,这时只剩下一个子查询，如果满足 RTE_SUBQUERY (SIMPLE)子查询提升的条件，那么还能进行子查询的提升，等价语句如下：

```
SELECT sname, 1, 2 FROM STUDENT WHERE STUDENT.sno = 1;
```

因为(SELECT 1,2) AS sc(sno, cno)是一个 empty-FROM 的子查询，因此最终连接操作被消除掉了。

要进行 RTE_VALUES 的提升需要满足的条件如下：

1）不能是外连接下的语句，不能是 UNION ALL 集合操作下的子句。

2）由于 RTE_VALUES 先提升到 targetList 中，提升之后 RTE_VALUES 类型的 RangeTblEntry 就没有什么用了，因此这个 RTE_VALUES 会被删除掉，这就需要确保 deleteOK == true，子查询才能进行提升。

3）RTE_VALUES 中只能有一个“元组”，如果包含多个的话，我们无法提升到 targetList（注：进一步思考一下，如果 RTE_VALUES 中有多个值，实际上可以通过 UNION 的方式形成多个提升到 targetList 中的查询树，也是可以“提升”的）。

4）VALUES 指定的值中如果有表达式，那么不能含有易失性函数，也不能是返回集合的表达式。

5）必须只能有一个 RTE_VALUES。

满足上述条件之后，就可以对 RTE_VALUES 进行提升，需要注意的是，由于层次关系是：顶层查询树（Query） -> RTE_SUBQUERY 类型的子查询 -> RTE_VALUES 类型的子查询，共三个层次，因此在提升时的函数调用顺序如图 3-8 所示。

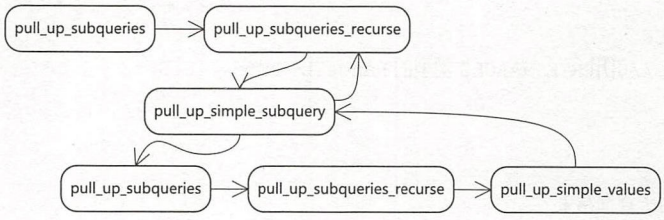


图 3-8　VALUES 类型的子查询提升函数调用关系

对于 RTE_VALUES 类型子查询的提升在 pull_up_simple_values 函数中实现，主要做了两部分工作。

1）将 VALUES 中对应的常量组织成 TargetEntry 的形式。

```
foreach(lc, values_list)
{
      tlist = lappend(tlist,
                  makeTargetEntry((Expr *) lfirst(lc),
                                  attrno,
                                  NULL,
                                  false));
      attrno++;
}
```

2）将 RTE_SUBQUERY 中引用了 VALUES 的所有变量全部替换（pullup_replace_vars 函数）。

在 RTE_VALUES 提升之后，RTE_VALUES 类型的 RangeTblEntry 就已经失去了原来的作用，因此需要被清除掉，这时候默认设置 root->hasDeletedRTEs = true。

在 RTE_VALUES 提升之后，原来的三层关系变成了两层关系，目前存在一个 RTE_

SUBQUERY(simple)类型的子查询，这个子查询还有一个特殊之处是没有范围表，也就是说它只在 targetList 中有值，这时查询的等价语句如下：

```
SELECT sname, sc.sno, sc.cno FROM STUDENT, (SELECT 1,2) AS sc(sno, cno) WHERE sc.sno =
STUDENT.sno;
```

这个子查询仍然能够提升，它是一个 empty-From 的子查询，具体的提升流程可以参照 pull_up_simple_subquery 函数。

另外再重点介绍一下 deleteOK 和 root->hasDeleteRTEs，这两个变量的引入主要是因为子查询提升的过程中处理了 VALUES 子查询和 empty-FROM 子查询，这两类子查询的特点是没有范围表，将这种子查询提升后，pull_up_subqueries_recurse 函数的返回值可能是一个 NULL 值，这就需要消除掉这个 NULL 值。

delete_ok 主要负责记录范围表中的这个 NULL 值是否能删除掉，在内连接的情况下，很明显是能够消除掉的，但是如果是外连接和半连接，即使是 NULL 值，也不一定能够消除掉。deleteOK 默认值是 false，在 pull_up_subqueries_recurse 函数对 FromExpr 和 JoinExpr 进行递归遍历时，根据连接的逻辑操作符来进行修正。

在 FromExpr 中对 delete_ok 的处理如下：
1．如果上层传递过来的是 delete_ok，我们就可以尝试让下层去消除。
2．如果是顶层的 FromExpr，那么尝试消除。
3．如果还没到最后一个（lnext(l) != NULL），那么尝试消除。
4．如果 fromlist 中有了不能消除的范围表（have_undeleted_child），对其他范围表可以尝试消除。
上面的 4 个部分不是互相独立的，它的主要思想是，如果上层已经指定了 delete_ok 的值是 true，那么实际上当前这个子节点整个都是能删除的，因此无须考虑 have_undeleted_child 的情况；如果 delete_ok 的值是 false，那么也就是说整个节点不能消除（例如整个节点可能在外连接之下），这时我们还可以尝试消除部分子节点，这时只要 have_undeleted_child 的值是 true，就证明已经有保留的节点，因此其他节点都可以尝试被消除，如果 have_undeleted_child 的值是 false 也没有关系，只要不是最后一个节点（lnext(l) != NULL），都可以尝试被删除。
还有一种情况，如果是顶层的 FromExpr，它的子节点就可以尝试全部被删除，因为 delete_ok 默认是 false，所以需要单独判断。

```
bool     sub_deletion_ok = (deletion_ok ||
                                          have_undeleted_child ||
                                          lnext(l) != NULL ||
                                          f == root->parse->jointree);
```

在 JoinExpr 中对 delete_ok 的处理如下：
1．如果是内连接，那么可以删除任何一个节点，但是不能同时删除两个节点。

```
        switch (j->jointype)
```

```
        {
            case JOIN_INNER:
                j->larg = pull_up_subqueries_recurse(root, j->larg,
                                                     lowest_outer_join,
                                                     lowest_nulling_outer_join,
                                                     NULL,
                                                     true);
                j->rarg = pull_up_subqueries_recurse(root, j->rarg,
                                                     lowest_outer_join,
                                                     lowest_nulling_outer_join,
                                                     NULL,
                                                     j->larg != NULL);
                break;
    }
2. 其他类型的连接，全部不能删除。
   ......
```

在提升子查询的过程中，通过设置 root->haveDeletedRTEs 来表示 Query->jointree 中有可以删除的 NULL 值节点，删除的具体工作则由 pull_up_subqueries_cleanup 函数来完成。

```
//如果设置了 root->hasDeletedRTEs 标志
//则调用 pull_up_subqueries_cleanup 函数删除 NULL 节点
if (root->hasDeletedRTEs)
    root->parse->jointree = (FromExpr *)
        pull_up_subqueries_cleanup((Node *) root->parse->jointree);
```

3.3 UNION ALL 优化

在子查询的提升过程中，对“简单”的 UNION ALL 集合操作进行了提升，但是并没有处理顶层的 UNION ALL 集合操作，这里通过 flatten_simple_union_all 函数对顶层的 UNION ALL 进行处理，目的是将 UNION ALL 这种集合操作的形式转换为 AppendRelInfo 的形式，例如下面的示例：

```
postgres=# EXPLAIN SELECT * FROM STUDENT WHERE sno = 1 UNION ALL SELECT * FROM STUDENT WHERE
sno = 2;
                        QUERY PLAN
-------------------------------------------------------------------------
 Append  (cost=0.00..2.19 rows=2 width=11)
   ->  Seq Scan on student  (cost=0.00..1.09 rows=1 width=11)
         Filter: (sno = 1)
```


非卖品，仅供非商业用途或交流学习使用

```
  ->  Seq Scan on student student_1  (cost=0.00..1.09 rows=1 width=11)
        Filter: (sno = 2)
(5 rows)
```

在没有进行 UNION ALL 优化之前，它的查询树的形式如图 3-9 所示。

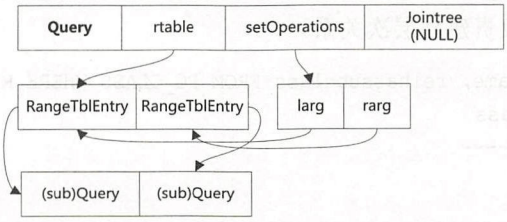


图 3-9　UNION 类型的集合操作优化前的查询树结构图

在进行 UNION ALL 优化之后，查询树的形式如图 3-10 所示。

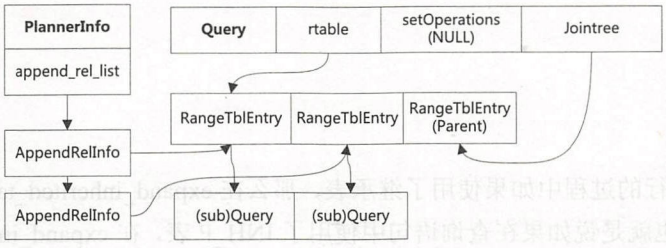


图 3-10　UNION 类型的集合操作优化后的查询树结构图

UNION ALL 操作向 AppendRelInfo 形式的转换的代码和处理 RTE_SUBQUERY(UNION)的代码是一致的，详情可以参考 RTE_SUBQUERY(UNION)的说明。

3.4　展开继承表

在子查询提升的过程中，对 UNION ALL 类型的操作符进行优化时已经涉及了继承表，因为 UNION ALL 和继承表有一定的相似性，最终都采用 AppendRelInfo 结构体来实现，UNION ALL 使用 AppendRelInfo 来封装子查询，提升子查询的层次，而继承表则用 AppendRelInfo 来记录子表。

继承表的使用对于用户是“透明”的，也就是说用户操作继承表的父表时，不需要了解继承表的细节，例如它有几个子表，每个子表的结构是什么样的，用户都无须知道，这些都由 PostgreSQL 数据库自动完成查询。

非卖品，仅供非商业用途或交流学习使用

如果用户创建了一个表 INH_P 作为父表，同时创建了 INH_C1 和 INH_C2 作为 INH_P 的子表，PostgreSQL 数据库在 PG_CLASS 系统表中表现的是 3 个表，不过 INH_P 在 SYS_CLASS 的 relhassubclass 列的值是 true，这就代表 INH_P 有子表（has_subclass 函数），而父表和子表的继承关系则记录在 PG_INHERITS 系统表中（find_inheritance_children 函数查找下一层子表，find_all_inheritors 函数负责建立层次关系）。

```
postgres=# SELECT relname, relhassubclass FROM PG_CLASS WHERE RELNAME = 'inh_p';
 relname | relhassubclass
---------+----------------
 inh_p   | t
(1 row)

postgres=# SELECT relname FROM PG_CLASS WHERE oid in (SELECT inhrelid FROM PG_INHERITS WHERE
inhparent = 32768);
 relname
---------
 inh_c1
 inh_c2
(2 rows)
```

在查询语句执行的过程中如果使用了继承表，那么在 expand_inherited_tables 函数之前，都还没有“展开”，也就是说如果在查询语句中使用了 INH_P 表，在 expand_inherited_tables 函数调用前，都是以一个普通表的方式存在的，expand_inherited_tables 负责将 INH_P 转换成继承表形式，继承表展开的流程如图 3-11 所示。

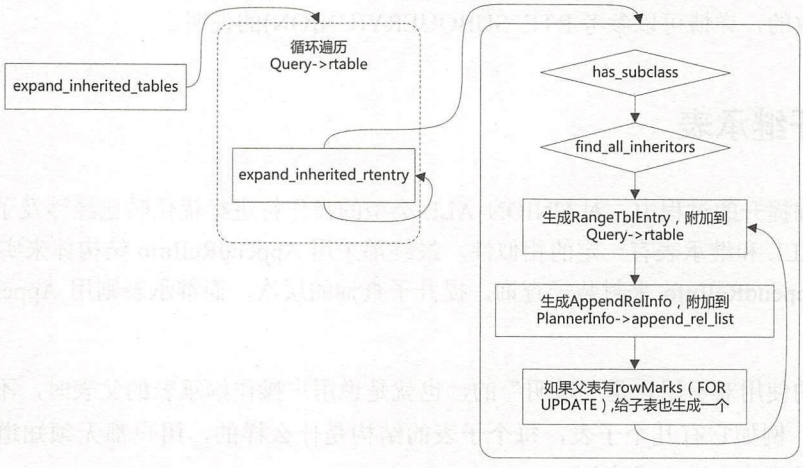


图 3-11　继承表展开流程图

非卖品，仅供非商业用途或交流学习使用

3.5　预处理表达式

预处理表达式（preprocess_expression 函数）是对查询树（Query）中的表达式进行规范整理的过程，包括对连接产生的别名 Var 进行替换、对常量表达式求值、对约束条件进行拉平、为子连接（SubLink）生成执行计划等。

3.5.1　连接 Var 的溯源

目前连接操作的投影都是基于连接结果的情况来生成的，我们在执行的过程中，需要将这些 Var 替换成连接操作的 LHS 或 RHS 的表上的 Var。

例如有 SQL 语句如下：

```
SELECT st.sno, sc.cname, sc.degree FROM STUDENT st, (COURSE INNER JOIN SCORE ON TRUE) sc WHERE st.sno = sc.sno
```

SQL 语句中涉及了 3 个表，分别是 STUDENT、SCORE、COURSE，在 Query->rtable 中，共有 4 个 RangeTblEntry，分别对应着 STUDENT、COURSE、SCORE、COURSE INNER JOIN SCORE，它的 Query->jointree 如图 3-12 所示。

示例的 SQL 语句中和 COURSE INNER JOIN SCORE 相关的列有：投影列中的 sc.cname 和 sc.degree，约束条件中的 sc.sno，这些列属性对应的值如图 3-13 所示，以 sc.cname（也就是 COURSE.cname）为例，它的 varno 是 4，varattno 是 2，代表它是第 4 个 RangeTblEntry 的第 2 列，也就是（COURSE INNER JOIN SCORE）结果的投影列的第 2 列。

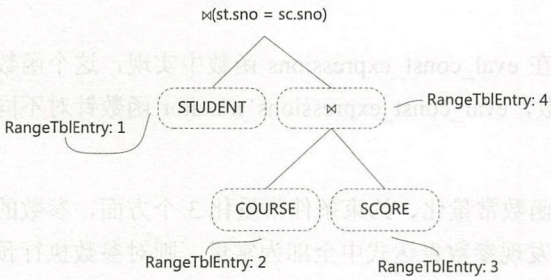


图 3-12　连接表的结构示意图

sc.cname	sc.degree	sc.sno
{VAR	{VAR	{VAR
:varno 4	:varno 4	:varno 4
:varattno 2	:varattno 6	:varattno 4
:vartype 1043	:vartype 23	:vartype 23
:vartypmod 14	:vartypmod -1	:vartypmod -1
:varcollid 100	:varcollid 0	:varcollid 0
:varlevelsup 0	:varlevelsup 0	:varlevelsup 0
:varnoold 4	:varnoold 4	:varnoold 4
:varoattno 2	:varoattno 6	:varoattno 4
:location 15	:location 25	:location 104
}	}	}

图 3-13　连接 Var 结构体的内存结构

COURSE INNER JOIN SCORE 在查询树中对应的投影列有 6 个 Var，保存在 RangeTblEntry->joinaliasvars 中，分别对应的是"cno"、"cname"、"tno"、"sno"、"cno"、"degree"，它们分别来自 COURSE 表和 SCORE 表，如图 3-14 所示，前 3 个 Var 的 varno 是 2，代表它们

非卖品，仅供非商业用途或交流学习使用

是来自 COURSE 的列，后 3 个 Var 的 varno 是 3，代表它们是来自 SCORE 的列。

COURSE.cno	COURSE.cname	COURSE.tno	SCORE.sno	SCORE.cno	SCORE.degree
{VAR :varno 2 :varattno 1 :vartype 23 :vartypmod -1 :varcollid 0 :varlevelsup 0 :varnoold 2 :varoattno 1 :location -1 }	{VAR :varno 2 :varattno 2 :vartype 1043 :vartypmod 14 :varcollid 100 :varlevelsup 0 :varnoold 2 :varoattno 2 :location -1 }	{VAR :varno 2 :varattno 3 :vartype 23 :vartypmod -1 :varcollid 0 :varlevelsup 0 :varnoold 2 :varoattno 3 :location -1 }	{VAR :varno 3 :varattno 1 :vartype 23 :vartypmod -1 :varcollid 0 :varlevelsup 0 :varnoold 3 :varoattno 1 :location -1 }	{VAR :varno 3 :varattno 2 :vartype 23 :vartypmod -1 :varcollid 0 :varlevelsup 0 :varnoold 3 :varoattno 2 :location -1 }	{VAR :varno 3 :varattno 3 :vartype 23 :vartypmod -1 :varcollid 0 :varlevelsup 0 :varnoold 3 :varoattno 3 :location -1 }

图 3-14　连接表中的 Var 的内存结构

flatten_join_alias_vars 函数的主要作用就是使用 joinaliasvars 中的 Var 替换那些引用连接结果的 Var，对示例语句而言就是用 COURSE.cname 替换 sc.cname、用 SCORE.degree 替换 sc.degree、用 SCORE.sno 替换 sc.sno，如图 3-15 所示。

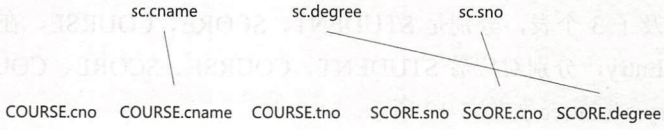


图 3-15　Var 替换对照图

```
//var->varattno - 1 代表在 joinaliasvars 中的第几列
newvar = (Node *) list_nth(rte->joinaliasvars, var->varattno - 1);
```

3.5.2　常量化简

对含有常量的表达预先进行求值，该功能在 eval_const_expressions 函数中实现，这个函数递归调用了 eval_const_expressions_mutator 函数，eval_const_expressions_mutator 函数针对不同的表达式做了不同的处理。

常量化简的主要的优化点有参数常量化、函数常量化、约束条件常量化 3 个方面，参数的常量化是通过遍历参数的表达式实现的，如果发现参数表达式中全部为常量，则对参数执行预先求值。从下面的示例可以看出，原来的 MAX(degree + (1+2))已经转变成了 MAX(degree+3)。

```
postgres=# EXPLAIN VERBOSE SELECT MAX(degree + (1+2)) FROM SCORE;
                        QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=40.60..40.61 rows=1 width=4)
   Output: max((degree + 3))
```


非卖品，仅供非商业用途或交流学习使用

```
  ->  Seq Scan on public.score  (cost=0.00..30.40 rows=2040 width=4)
        Output: sno, cno, degree
(4 rows)
```

函数常量化是通过 simplify_function 函数来实现的，它如果发现所有的参数都是常量，则尝试预先获取函数的执行结果，并将结果常量化。从下面的示例可以看出，int4ge 函数是负责比较两个 int4 类型的，在执行计划里直接给出了 int4ge 函数的结果。

```
postgres=# EXPLAIN VERBOSE SELECT int4ge(1,3);
                QUERY PLAN
------------------------------------------
 Result  (cost=0.00..0.01 rows=1 width=1)
   Output: false
(2 rows)
```

实际上这里的常量化值进行了比较基础的分析，比如我们将参数的常量化略作调整，这种优化就无法进行了。如下面的示例所示，由于所有操作符是左结合的，因此导致投影表达式中的每个操作符都有非常量，这就导致了无法进行常量化。

```
postgres=# EXPLAIN VERBOSE SELECT MAX(degree + 1 + 2) FROM SCORE;
                          QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=45.70..45.71 rows=1 width=4)
   Output: max(((degree + 1) + 2))
   ->  Seq Scan on public.score  (cost=0.00..30.40 rows=2040 width=4)
         Output: sno, cno, degree
(4 rows)
```

3.5.3　谓词规范

谓词规范是对约束条件的形式进行规范化，主要包括 3 个功能。

- 简化约束条件。
- 对树状的约束条件拉平，将形如 A OR (B OR C)形式的约束条件转换为 OR(A,B,C)形式。
- 提取公共项，将形如(A AND B) OR (A AND C)转换为 A AND (B OR C)。

3.5.3.1　谓词规约

例如有这样一个 SQL 语句：SELECT * FROM STUDENT WHERE NULL OR FALSE OR sno = 1，由于是 OR 操作，约束条件中的 NULL 和 FALSE 是可以忽略掉的，我们看一下执行计划：

非卖品，仅供非商业用途或交流学习使用

```
postgres=# EXPLAIN SELECT * FROM STUDENT WHERE NULL OR FALSE OR sno = 1;
                    QUERY PLAN
-----------------------------------------------------
 Seq Scan on student  (cost=0.00..1.09 rows=1 width=11)
   Filter: (sno = 1)
(2 rows)
```

对于 AND 操作，如果涉及了 NULL 或 FALSE，则实际上代表整个约束条件可以规约为 FALSE，例如对于 SQL 语句 SELECT * FROM STUDENT WHERE NULL AND FALSE AND sno = 1，约束条件 NULL AND FALSE AND sno = 1 的值最终是 FALSE，这个语句的执行计划如下：

```
postgres=# EXPLAIN SELECT * FROM STUDENT WHERE NULL AND FALSE AND sno = 1;
              QUERY PLAN
-----------------------------------------
 Result  (cost=0.00..0.00 rows=0 width=46)
   One-Time Filter: false
(2 rows)
```

这部分的实现在 find_duplicate_ors 函数中。

```
//对 OR 中的 NULL 或 FALSE 的处理
if (arg && IsA(arg, Const))
{
   Const      *carg = (Const *) arg;

   //如果发现了 NULL 或者 FALSE，则跳过，不把这个常量参数记录下来
   if (carg->constisnull || !DatumGetBool(carg->constvalue))
      continue;

   //否则，这个常量是真值，则这个约束条件规约为 TRUE
   return arg;
}

//对 AND 中的 NULL 或 FALSE 的处理
if (arg && IsA(arg, Const))
{
   Const      *carg = (Const *) arg;

   //如果常量是真值，则跳过，不把这个常量参数记录下来
   if (!carg->constisnull && DatumGetBool(carg->constvalue))
      continue;
   //否则，代表这个常量是 NULL 或者 FALSE，这个约束条件规约为 FALSE
```


非卖品，仅供非商业用途或交流学习使用

```
    return (Expr *) makeBoolConst(false, false);
}
```

3.5.3.2　谓词拉平

先来看一下约束条件在查询树的表现形式，例如有 SQL 语句：SELECT * FROM STUDENT WHERE sno=1 OR (sno=2 OR (sno=3 OR sno=4))，它的约束条件的组织形式如图 3-16 所示，是一个树状结构，谓词规范需要将这个约束条件拉平，我们来看一下这个语句的执行计划：

```
postgres=# EXPLAIN SELECT * FROM STUDENT WHERE sno=1 OR (sno=2 OR (sno=3 OR sno=4));
                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on student  (cost=0.00..1.14 rows=3 width=11)
   Filter: ((sno = 1) OR (sno = 2) OR (sno = 3) OR (sno = 4))
(2 rows)
```

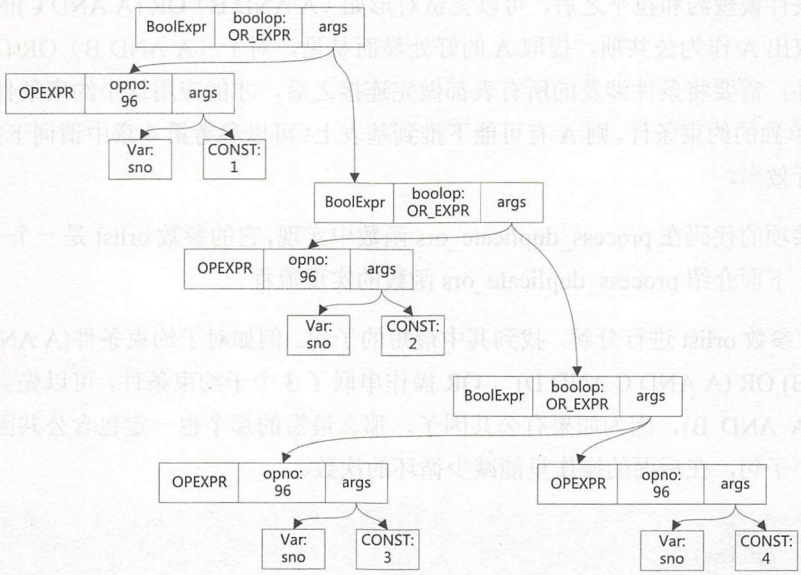


图 3-16　操作符表达式内存结构图

从执行计划可以看出，这个约束条件已经拉平了，所有的 OR 子句都是在同一个层次上。

对 OR 的拉平是在 pull_ors 函数中实现的，它的主要功能是通过递归的方式找到 BoolExpr，并且将 BoolExpr 中的 args 附加到一个链表里。

```
foreach(arg, orlist)
{
    Node      *subexpr = (Node *) lfirst(arg);
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
    //如果是 OR 子句，则递归处理
    if (or_clause(subexpr))
        out_list = list_concat(out_list,
                               pull_ors(((BoolExpr *) subexpr)->args));
    else
        //否则将表达式附加到链表返回
        out_list = lappend(out_list, subexpr);
}
```

对于含有 AND 操作符的子句，则调用 pull_ands 函数进行拉平，实现的方式和 pull_ors 相似，都是递归处理 BoolExpr 中的 args。

3.5.3.3 提取公共项

在约束条件被规约和拉平之后，可以尝试对形如（A AND B）OR (A AND C)的约束条件进行优化，提取出 A 作为公共项，提取 A 的好处显而易见。对于（A AND B）OR (A AND C)这样的约束条件，需要将条件涉及的所有表都做完连接之后，才能应用这个约束条件。而如果提取出 A 作为单独的约束条件，则 A 有可能下推到基表上（可以参考第 4 章中谓词下推的部分），从而提高执行效率。

提取公共项的代码在 process_duplicate_ors 函数中实现，它的参数 orlist 是一个 AND-of-ORs 形式的链表，下面介绍 process_duplicate_ors 函数的实现流程。

首先，对参数 orlist 进行分解，找到其中最短的子句。例如对于约束条件(A AND B AND C) OR (A AND B) OR (A AND C AND D) ，OR 操作串联了 3 个子约束条件，可以先尝试找到其中最短的一个(A AND B)，因为如果有公共因子，那么最短的那个也一定包含公共因子，通过找到最短的那个子句，在后面的操作里能减少循环的次数。

```
//找最短
foreach(temp, orlist)
{
    Expr   *clause = (Expr *) lfirst(temp);

    //查看 AND 约束条件的长度
    if (and_clause((Node *) clause))
    {
        List   *subclauses = ((BoolExpr *) clause)->args;
        int           nclauses = list_length(subclauses); //获得长度

        //比较长度，记录比较短的约束条件
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
        if (reference == NIL || nclauses < num_subclauses)
        {
            reference = subclauses;
            num_subclauses = nclauses;
        }
    }
    else
    {
        //如果有不是 AND 类型的约束条件，全部记录下来
        //这里可能是带有表达式的约束条件或者单个的约束条件
        //例 1: (A AND B) OR A
        //例 2: (A IS NULL AND B) OR A IS NULL
        reference = list_make1(clause);
        break;
    }
}
```

然后，以最短的约束条件为依据，提取公共项，对于(A AND B)这样的最短项，可以先查看一下 3 个 AND 子句中是否都包含 A。

- (A AND B AND C)：包含 A。
- (A AND B)：包含 A。
- (A AND C AND D)：包含 A。

如果都包含 A，那么 A 就是公共项中的一员，再参照 3 个 AND 子句看是否都包含 B。

- (A AND B AND C) ：包含 B。
- (A AND B) ：包含 B。
- (A AND C AND D) ：不包含 B。

因为(A AND C AND D)不包含 B，所以 B 不是公共项中的一员，最终可以得出公共项为 A，在得到公共项之后，就可以开始提取公共项，提取过程如下：

```
//提取公共项
neworlist = NIL;
foreach(temp, orlist)
{
   Expr      *clause = (Expr *) lfirst(temp);

   if (and_clause((Node *) clause))
   {
      List      *subclauses = ((BoolExpr *) clause)->args;
```

版权所有，严禁传播，违者自负法律责任！


```
        //winners 是公共项，提取出去
        subclauses = list_difference(subclauses, winners);
        if (subclauses != NIL)
        {
            //提取之后，将剩余的项生成一个新的约束条件，保存到 neworlist 中
            if (list_length(subclauses) == 1)
                neworlist = lappend(neworlist, linitial(subclauses));
            else
                neworlist = lappend(neworlist, make_andclause(subclauses));
        }
        else
        {
            //如果提取之后没有剩余项，例如(A AND B AND C) OR (A AND B)
            neworlist = NIL;    /* degenerate case, see above */
            break;
        }
    }
    else
    {
        if (!list_member(winners, clause))
            neworlist = lappend(neworlist, clause);
        else
        {
            //如果提取之后没有剩余项，例如(A AND B) OR (A)
            neworlist = NIL;    /* degenerate case, see above */
            break;
        }
    }
}
```

需要注意的是，对于(A AND B) OR (A)这样的约束条件可以规约成(A)，同理，对于(A AND B AND C) OR (A AND B)这种类型的约束条件可以规约成(A AND B)。

```
postgres=# EXPLAIN SELECT * FROM STUDENT WHERE (sno = 1 AND sname = 'zs') OR sno = 1;
                    QUERY PLAN
-----------------------------------------------------
 Seq Scan on student  (cost=0.00..1.09 rows=1 width=11)
   Filter: (sno = 1)
(2 rows)
```

版权所有，严禁传播，违者自负法律责任！

3.5.4　子连接处理

在子连接提升的过程中，我们只尝试提升了 EXISTS 类型和 ANY 类型的子连接，而且要求 EXISTS 类型的子连接必须是相关子连接，要求 ANY 类型的子连接必须是非相关子连接，对于其他类型的子连接都没有进行处理。

我们已经分析过子连接和子查询的区别在于子连接是处在表达式中的子查询，如果不能对其进行提升形成连接关系，那么就必须为其生成一个子执行计划，父子执行计划之间可以通过 Param 结构体互相通信。

子执行计划的生成和父执行计划的生成别无二致，因此不是我们关注的重点，但是在生成子执行计划的 make_subplan 函数中还做了一些特别的优化，需要特别关注一下。

一个 EXISTS 类型的子连接，它通常只需要关心是“存在”还是“不存在”，至于是以什么样的形式“存在”和“不存在”它并不关心，因此可以对子连接进行简化，消减掉一些对执行结果没有影响的操作，在子连接提升的时候也做过同样的判断，这里不再赘述。

另外子执行计划存在的形式也不完全相同，它还分成了相关子执行计划和非相关子执行计划。如果是非相关子连接，那么它的值通常是固定的，因此可以为其值生成一个 Param，这样在父执行计划要获取这个值的时候才会执行子执行计划，也就是说会在 ExecEvalParam 函数中对这个子执行计划求解，并且将获取的值记录在 Param 中，当查询执行器再次对 Param 求解的时候如果发现要获取的值已经存在，则直接使用以前获取的值。从下面的示例可以看出，非相关子连接产生了一个 Param 参数。

```
postgres=# EXPLAIN VERBOSE SELECT * FROM TEST_A WHERE EXISTS (SELECT a FROM TEST_B);
                        QUERY PLAN
-----------------------------------------------------------------------
 Result  (cost=27.01..84.57 rows=256 width=16)
   Output: test_a.a, test_a.b, test_a.c, test_a.d
   One-Time Filter: $0
   InitPlan 1 (returns $0)  --Param
     ->  Seq Scan on public.test_b  (cost=0.00..54.02 rows=2 width=0)
   ->  Seq Scan on public.test_a  (cost=27.01..84.57 rows=256 width=16)
         Output: test_a.a, test_a.b, test_a.c, test_a.d
(7 rows)

postgres=# EXPLAIN VERBOSE SELECT (SELECT a FROM TEST_B LIMIT 1), a FROM TEST_A;;
                        QUERY PLAN
-----------------------------------------------------------------------
```

版权所有，严禁传播，违者自负法律责任！


```
 Seq Scan on public.test_a  (cost=27.01..84.57 rows=256 width=8)
   Output: $0, test_a.a
   InitPlan 1 (returns $0)  --Param
     -> Limit  (cost=0.00..27.01 rows=1 width=4)
          Output: test_b.a
          -> Seq Scan on public.test_b  (cost=0.00..54.02 rows=2 width=4)
               Output: test_b.a
(7 rows)
```

而对相关子连接，则直接返回它的子执行计划，但实际上它们也是通过 Param 来进行通信的，只不过它的 Param 是在生成子执行计划的过程中生成的，而且如果父查询计划中使用了子执行计划的 Var，也要将父执行计划中的 Var 调整成 Param。下面是一个相关子连接的例子，示例的执行计划中没有打印相关的 Param，但是通过执行计划可以看出，非相关子连接中的子执行计划标记为 InitPlan，而相关子连接中的子执行计划标记为 SubPlan。

```
postgres=# EXPLAIN VERBOSE SELECT * FROM TEST_A WHERE a > ANY(SELECT a FROM TEST_B WHERE b >
TEST_A.b);
                          QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on public.test_a  (cost=0.00..6973.72 rows=128 width=16)
   Output: test_a.a, test_a.b, test_a.c, test_a.d
   Filter: (SubPlan 1)
   SubPlan 1
     -> Seq Scan on public.test_b  (cost=0.00..54.02 rows=1 width=4)
          Output: test_b.a
          Filter: (test_b.b > test_a.b)
(7 rows)
```

3.6 处理 HAVING 子句

在 Having 子句中，有些约束条件是可以转变为过滤条件的，这里对 Having 子句中的约束条件进行了拆分。从下面的示例可以看出，cno > 0 这个约束条件已经变成了 SCORE 表扫描路径上的过滤条件，而 SUM(degree) > 100 这个约束条件则保留在了原来的位置，这部分源代码的逻辑比较简单，有兴趣的读者可自行分析。

```
postgres=# EXPLAIN SELECT SUM(degree), sno, cno FROM SCORE WHERE sno > 0 GROUP BY sno, cno
HAVING SUM(degree) > 100 AND cno > 0;
                      QUERY PLAN
---------------------------------------------------------------
```

版权所有，严禁传播，违者自负法律责任！


```
 HashAggregate  (cost=42.87..44.63 rows=47 width=16)
   Group Key: sno, cno
   Filter: (sum(degree) > 100)
   ->  Seq Scan on score  (cost=0.00..40.60 rows=227 width=12)
         Filter: ((sno > 0) AND (cno > 0))
(5 rows)
```

3.7　Group By 键值消除

Group By 子句的实现需要借助排序或者 Hash 来实现，如果能减少 Group By 后面的字段，就能降低排序或者 Hash 带来的损耗。

对于一个有主键（Primary Key）的表，如果 Group By 的字段包含了主键的所有键值，实际上这个主键已经能够表示当前的结果就是符合分组的，因此可以将主键之外的字段去掉。例如对于 SQL 语句 SELECT * FROM STUDENT GROUP BY sno, sname,ssex，因为 sno 属性上有一个主键索引，所以 Group By 子句中的 sname 和 ssex 可以被去除掉。

```
postgres=# EXPLAIN SELECT * FROM STUDENT GROUP BY sno, sname, ssex;
                        QUERY PLAN
-------------------------------------------------------------
 HashAggregate  (cost=1.09..1.16 rows=7 width=11)
   Group Key: sno
   ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
(3 rows)
```

上面的示例是单表的扫描路径，连接路径也可以用同样的方式简化 Group By 子句。无论是连接操作的 LHS 还是 RHS，只要 Group By 包含了表上的全部主键，那么主键键值之外的 Group By 字段就能被消除掉。

```
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN COURSE ON TRUE GROUP BY
sno,sname,ssex,cno,cname;
                              QUERY PLAN
-----------------------------------------------------------------------------
 HashAggregate  (cost=159.57..236.57 rows=7700 width=57)
   Group Key: student.sno, course.cno
   ->  Nested Loop Left Join  (cost=0.00..121.07 rows=7700 width=57)
         ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
         ->  Materialize  (cost=0.00..26.50 rows=1100 width=46)
               ->  Seq Scan on course  (cost=0.00..21.00 rows=1100 width=46)
(6 rows)
```

版权所有，严禁传播，违者自负法律责任！

消除的具体过程是在 remove_useless_groupby_columns 函数中实现的，主要流程如图 3-17 所示。

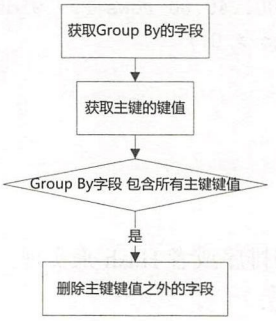


图 3-17　Group By 键值消除流程图

3.8　外连接消除

在查询优化的过程中，很多时间都是在和外连接、（反）半连接做斗争。例如对约束条件进行下推（谓词下推）时，如果连接操作是外连接，那么有些约束条件下推会受到阻碍，再例如连接顺序交换，内连接的表之间的连接顺序交换比较灵活，而外连接不能随意地交换连接表的顺序，因此，如果能将外连接转换成内连接，查询优化的过程就会大大地简化。

下面先来看一下内连接和外连接的区别，例如两个表 STUDENT 和 SCORE 的数据如下，其中有一个学生 lisi 没有成绩。

```
postgres=# SELECT * FROM STUDENT;
 sno  |  sname    | ssex
------+----------+------
   1  | zhangsan |    1
   2  | lisi     |    1
(2 rows)

postgres=# SELECT * FROM SCORE;
 sno  | cno | degree
------+-----+--------
   1  |  1  |    36
(1 row)
```

如果要查询学生的姓名和成绩，可以对 STUDENT 和 SCORE 做连接，从查询结果可以看

版权所有，严禁传播，违者自负法律责任！

出，内连接只显示了有成绩的学生，而外连接则对没有成绩的学生补了 NULL 值，也就是说这个外连接是不能转换成内连接的。

```
postgres=# SELECT * FROM STUDENT LEFT JOIN SCORE ON STUDENT.sno = SCORE.sno;
 sno  |  sname    | ssex | sno | cno | degree
------+----------+------+-----+-----+--------
   1  | zhangsan |   1  |  1  |  1  |    36
   2  | lisi     |   1  |     |     |
(2 rows)

postgres=# SELECT * FROM STUDENT INNER JOIN SCORE ON STUDENT.sno = SCORE.sno;
 sno  |  sname    | ssex | sno | cno | degree
------+----------+------+-----+-----+--------
   1  | zhangsan |   1  |  1  |  1  |    36
(1 row)
```

假如我们再增加一个 WHERE 条件，形成下面这样的语句，内连接的查询结果就和外连接的查询结果相同了。

```
postgres=# SELECT * FROM STUDENT LEFT JOIN SCORE ON STUDENT.sno = SCORE.sno WHERE cno IS NOT
NULL;
 sno  |  sname    | ssex | sno | cno | degree
------+----------+------+-----+-----+--------
   1  | zhangsan |   1  |  1  |  1  |    36
(1 row)

postgres=# SELECT * FROM STUDENT INNER JOIN SCORE ON STUDENT.sno = SCORE.sno WHERE cno IS
NOT NULL;
 sno  |  sname    | ssex | sno | cno | degree
------+----------+------+-----+-----+--------
   1  | zhangsan |   1  |  1  |  1  |    36
(1 row)
```

“WHERE cno IS NOT NULL”这样的条件可以让外连接和内连接的结果相同，因为这个约束条件是“严格（strict）”的。

“严格（strict）”的精确定义是对于一个函数、操作符或者表达式，如果输入参数是 NULL 值，那么输出也一定是 NULL 值，就可以说这个函数、操作符或者表达式是严格的；但是，宽泛地说，对于函数、操作符或者表达式，如果输入参数是 NULL 值，输出结果是 NULL 值或者 FALSE，那么就认为这个函数或者操作符是严格的。如果在约束条件里有这种严格的操作符、

函数或者表达式，由于输入是 NULL 值，输出是 NULL 值或者 FALSE，那么对于含有 NULL 值的元组就会被过滤掉。

“WHERE cno IS NOT NULL”这样的约束条件，如果输入的 cno 是 NULL 值，这个约束条件返回的是 FALSE，也满足了宽泛的“严格”定义，而且 cno 又处于左连接的 Nullable-side，对于补充的 NULL 值又能起到过滤的作用，因此增加它可以导致内连接和外连接的查询结果相同。

综上，就可以得出外连接能够被消除的条件。

- **上层**有“严格”的约束条件。
- 约束条件中引用了 Nullable-side 的表。

需要注意的是，消除条件里的“上层”两个字，所谓的上层是指这个约束条件不是当前的连接条件，而是上层的**连接条件**或者**过滤条件**，在第 4 章中介绍谓词下推时会详细地介绍连接条件和过滤条件，目前我们可以粗略地认为连接条件是 ON 关键字后面的约束条件，而过滤条件是 WHERE 关键字后面的约束条件。

例如有 SQL 语句：

```
SELECT * FROM STUDENT LEFT JOIN (SCORE LEFT JOIN COURSE ON TRUE) ON COURSE.cno IS NOT NULL;
```

其中约束条件（或者连接条件）“ON COURSE.cno IS NOT NULL”处在(SCORE LEFT JOIN COURSE ON TRUE)的上层，它能对(SCORE LEFT JOIN COURSE ON TRUE)这个外连接的消除起作用，但是不能对 STUDENT **LEFT JOIN** (SCORE LEFT JOIN COURSE ON TRUE)的消除起到作用，它们的层次关系如图 3-18 所示，例句的执行计划如下：

```
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN (SCORE LEFT JOIN COURSE ON TRUE) ON COURSE.cno
IS NOT NULL;
                          QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..132.48 rows=7658 width=69)
   ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
   ->  Materialize  (cost=0.00..38.42 rows=1094 width=58)
         ->  Nested Loop  (cost=0.00..32.95 rows=1094 width=58)
               ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
               ->  Seq Scan on course  (cost=0.00..21.00 rows=1094 width=46)
                     Filter: (cno IS NOT NULL)
(7 rows)
```

从执行计划可以看出(SCORE LEFT JOIN COURSE ON TRUE)的左连接已经被消除了，但

是 STUDENT **LEFT JOIN** (SCORE LEFT JOIN COURSE ON TRUE)的左连接仍然存在。

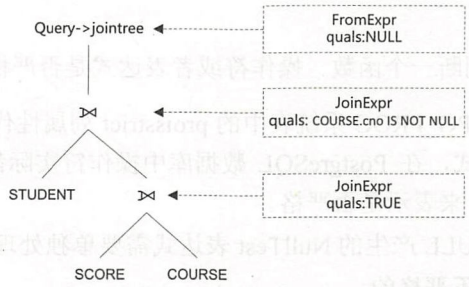


图 3-18　查询树中的连接关系示意图

再看另一个例句：

```
SELECT * FROM STUDENT LEFT JOIN (SCORE LEFT JOIN COURSE ON TRUE) ON TRUE WHERE COURSE.cno
IS NOT NULL;
```

它的结构图如图 3-19 所示，可以看到约束条件（或者过滤条件）“WHERE COURSE.cno IS NOT NULL”可以作用于顶层的连接，查看它的查询计划可以看出，语句中的左连接都被消除了。

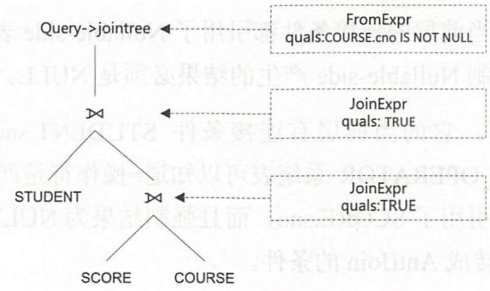


图 3-19　查询树中的连接关系示意图

```
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN (SCORE LEFT JOIN COURSE ON TRUE) ON TRUE
WHERE COURSE.cno IS NOT NULL;
                        QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=0.00..118.89 rows=7658 width=69)
   ->  Seq Scan on course  (cost=0.00..21.00 rows=1094 width=46)
         Filter: (cno IS NOT NULL)
   ->  Materialize  (cost=0.00..2.19 rows=7 width=23)
         ->  Nested Loop  (cost=0.00..2.15 rows=7 width=23)
```

```
        ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
        ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
(7 rows)
```

可以通过如下方法来判断一个函数、操作符或者表达式是否严格。

- 对于函数而言，在 PG_PROC 系统表中的 proisstrict 列属性代表了当前函数是否严格。
- 如果是操作符表达式，在 PostgreSQL 数据库中操作符实际都转成了对应的函数，因此也可以用 proisstrict 来表示是否严格。
- 对基于 IS [NOT] NULL 产生的 NullTest 表达式需要单独处理，其中 IS NOT NULL 是严格的，IS NULL 是不严格的。

如果给定一个表达式，那么可以对表达式进行递归遍历，如果满足上面的 3 种情况，那么这个表达式也是严格的。

而 IS NULL 这样不严格的表达式对我们也是有用的，例如，对于左连接而言，Nonnullable-side 没有连接上的元组会在 Nullable-side 补 NULL 值显示出来，而所谓的“没有连接上的元组”，恰好是 Anti Join 所需要的，因此就带来了将左连接（LeftJoin）转换成反连接（AntiJoin）的可能性，这种可能性的前提是：

- 当前层次的连接条件必须是严格的。
- 上层的约束条件和当前层的连接条件都引用了 Nullable-side 表的同一列。
- 上层的约束条件强制 Nullable-side 产生的结果必须是 NULL。

来看一个这样的示例，它的当前层有连接条件 STUDENT.sno = SCORE.sno，通过查 PG_PROC 系统表以及 PG_OPERATOR 系统表可以知道=操作符是严格的，它的上层的约束条件 SCORE.sno IS NULL 也引用了 SCORE.sno，而且强制结果为 NULL，而且 SCORE 表处在左连接的 RHS，因此它符合转成 AntiJoin 的条件。

```
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN SCORE ON STUDENT.sno = SCORE.sno WHERE
SCORE.sno IS NULL;
                        QUERY PLAN
------------------------------------------------------------------
 Nested Loop Anti Join  (cost=0.00..2.19 rows=6 width=23)
   Join Filter: (student.sno = score.sno)
   ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
   ->  Materialize  (cost=0.00..1.01 rows=1 width=12)
         ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
(5 rows)
```

从示例中可以看出，LeftJoin 被转换成了 AntiJoin，这是因为：连接条件 STUDENT.sno = SCORE.sno 是严格的，这保证了在 Nullable-side 的表中如果本身就含有 NULL 值，这些元组会被连接条件筛选掉，另外，约束条件 SCORE.sno IS NULL 是上层的不严格的约束条件，这就保证了在外连接操作之后，约束条件 SCORE.sno IS NULL 会把 Nullable-side 补的 NULL 值的元组保留下来了，这样的操作结果和 Anti Join 的结果应该是一致的。也就是说通过连接条件 STUDENT.sno = SCORE.sno 筛选掉了表中本来就有的 NULL 值，通过 SCORE.sno IS NULL 保留了外连接补的 NULL 值。

PostgreSQL 数据库通过 reduce_outer_joins 函数来消除外连接的同时，还做了一个“额外”的工作：将所有的右外连接都转换为左外连接，这样的工作在当前好像还看不到什么好处，但是当逻辑优化进入谓词下推、连接顺序交换的阶段时，查询优化就能够少处理一种情况，这会极大地简化源代码的逻辑。下面来看一下 reduce_outer_joins 函数的实现流程。

reduce_outer_joins 函数分成了两个步骤，第一个步骤是先做一个预检，查看一下查询树中是否存在外连接，外连接的层次结构是什么样的，这个层次结构通过 reduce_outer_joins_state 结构体来记录。

```
typedef struct reduce_outer_joins_state
{
    Relids    relids;           // 当前层次及下层引用了哪些表
    bool      contains_outer;   // 下层是否有外连接
    List    *sub_states;        // 下层的 reduce_outer_joins_state，树状结构
} reduce_outer_joins_state;
```

预检操作在一个叫作 reduce_outer_joins_pass1 的函数中完成，它递归的 Query->jointree，对其中的 RangeTblRef、FromExpr、JoinExpr 进行处理。

- 对于 RangeTblRef 直接记录它的 rtindex 返回给上层。
- 对于 FromExpr，默认当前层次是内连接，递归遍历 FromExpr->fromlist，对下层的连接进行预检，查看下层是否包含外连接。
- 对于 JoinExpr，如果是外连接，先设置 contains_outer 变量为 true，然后递归遍历 JoinExpr->larg 和 JoinExpr->rarg，查看下层是否包含外连接。

对于(STUDENT INNER JOIN SCORE) LEFT JOIN (COURSE LEFT JOIN TEACHER)这种形式，在经过 reduce_outer_joins_pass1 函数处理后，返回的 reduce_outer_joins_state 树状图如图 3-20 所示，通过结构图可以看出，contains_outer 的值为 false 的子树可以被剪掉，这样就能简化 reduce_outer_joins_pass2 函数的处理流程。

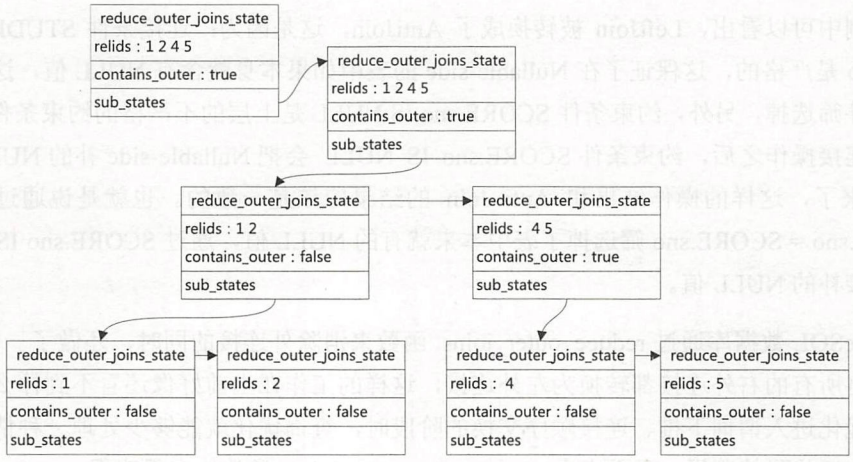


图 3-20　外连接消除结构内存示意图

reduce_outer_joins_pass2 函数开始对外连接消除，它有 3 个比较重要的参数，这 3 个参数的说明如表 3-9 所示。

表 3-9　reduce_outer_joins_pass2 函数参数说明

参数名	参数类型	描述
nonnullable_rels	[IN] Relids	收集严格约束条件所涉及的表的rtindex，收集工作由find_nonnullable_rels函数实现，函数的参数是FromExpr->quals或者JoinExpr->quals
nonnullable_vars	[IN]List*	收集严格约束条件中的Var，收集工作由find_nonnullable_vars函数实现，函数的参数是FromExpr->quals或者JoinExpr->quals
forced_null_vars	[IN] List *	收集不严格约束条件中的Var，收集工作由find_forced_null_vars函数实现，函数的参数是FromExpr->quals或者JoinExpr->quals

例如有约束条件 STUDENT.sno = 1 OR STUDENT.sno = 'zhangsan'，收集对应的 3 个变量的值如下。

- nonnullable_rels：STUDENT 表的 rtindex。
- nonnullable_vars：STUDENT.sno 对应的 Var。
- forced_null_vars：NULL。

再例如对约束条件(STUDENT.sno = 1 OR SCORE.sno = 2) AND STUDENT.sname IS NULL，通过收集 3 个变量的值如下。

- nonnullable_rels：无。
- nonnullable_vars：NULL。

- forced_null_vars：STUDENT.sname 对应的 Var。

在处理 FromExpr 时，会收集 FromExpr->quals 中的当前层变量，然后加入从函数参数传递过来的 3 个参数变量中，然后将合并的值传递给“下层”使用。

```
//收集严格约束条件涉及的表的 rtindex
pass_nonnullable_rels = find_nonnullable_rels(f->quals);
//和参数传进来的集合合并
pass_nonnullable_rels = bms_add_members(pass_nonnullable_rels, nonnullable_rels);

//收集严格约束条件涉及的 Var
pass_nonnullable_vars = find_nonnullable_vars(f->quals);
//和参数传进来的列表合并
pass_nonnullable_vars = list_concat(pass_nonnullable_vars, nonnullable_vars);

//收集非严格约束条件涉及的 Var
pass_forced_null_vars = find_forced_null_vars(f->quals);
//和参数传进来的列表合并
pass_forced_null_vars = list_concat(pass_forced_null_vars, forced_null_vars);

//如果下层有外连接，上面这 3 个变量会传递下去
```

在处理 JoinExpr 时，并非所有变量都会传递下去，传递的规则如下。

- 如果外连接在左子树：
  - 如果是内连接或者半连接，则传递参数变量+当前层变量给下层。
  - 如果是左连接或者反半连接，则传递参数变量给下层。
  - 如果是全连接，不传递变量给下层。
- 如果外连接在右子树：
  - 如果不是全连接，传递参数变量+当前层变量给下层。
  - 如果是全连接，不传递变量给下层。

外连接消除的过程就是判断 nonnullable_rels 变量和 Nullable-side 的表有没有交集的过程。

```
switch (jointype)
{
   case JOIN_INNER:
      break;
   case JOIN_LEFT:
      //左连接，nonnullable_rels 和 Nullable-side 的表有交集
      if (bms_overlap(nonnullable_rels, right_state->relids))
```

```
            jointype = JOIN_INNER;
        break;
    case JOIN_RIGHT:
        //右连接，nonnullable_rels 和 Nullable-side 的表有交集
        if (bms_overlap(nonnullable_rels, left_state->relids))
            jointype = JOIN_INNER;
        break;
    case JOIN_FULL:
        //全连接的两端都可以被认为是 Nullable-side
        //如果 nonnullable_rels 和 LHS&RHS 都有交集，转变为内连接
        //如果 nonnullable_rels 和 LHS 有交集，转变为右连接
        //如果 nonnullable_rels 和 RHS 有交集，转变为左连接
        if (bms_overlap(nonnullable_rels, left_state->relids))
        {
            if (bms_overlap(nonnullable_rels, right_state->relids))
                jointype = JOIN_INNER;
            else
                jointype = JOIN_LEFT;
        }
        else
        {
            if (bms_overlap(nonnullable_rels, right_state->relids))
                jointype = JOIN_RIGHT;
        }
        break;

        ……
}
```

左连接转换为反半连接的判断条件如下。

```
if (jointype == JOIN_LEFT)
{
    List      *overlap;

    //当前层有严格的约束条件
    local_nonnullable_vars = find_nonnullable_vars(j->quals);
    computed_local_nonnullable_vars = true;

    //当前层的约束条件和上层传递进来的非严格条件有交集
    overlap = list_intersection(local_nonnullable_vars,
                                forced_null_vars);
```

```
    //交集中的 Var，引用了 Nullable-side 的表，则转换为反半连接
    if (overlap != NIL &&
        bms_overlap(pull_varnos((Node *) overlap),
                    right_state->relids))
        jointype = JOIN_ANTI;
}
```

3.9　grouping_planner 的说明

在进行了外连接消除之后，查询优化器应该进入 grouping_planner 函数。这个函数做了两件事情。

- 对 SPJ 操作进行优化，例如进行谓词下推、计算路径代价、产生最优路径等，这部分主要在 query_planner 函数中完成。
- 在 SPJ 操作的最优路径的基础上，叠加 Non-SPJ 操作路径，例如生成基于 Group By、LIMIT、ORDER BY 的执行路径。

grouping_planner 函数中关于 Non-SPJ 优化的部分会在第 9 章进行分析，但是这里需要注意两个问题。

首先，Limit 子句对启动代价有比较大的影响，因此由 Limit 子句产生的 tuple_fraction 变量在代价计算的过程中会发挥很大的作用。

```
//如果语句中指定了 limit/offset，可以手动调整 tuple_fraction，对启动代价计算很重要
//这里同时也处理了常量 limit/offset，获得的常量 limit/offset 给 create_limit_path 函数使用
if (parse->limitCount || parse->limitOffset)
{
    tuple_fraction = preprocess_limit(root, tuple_fraction,
                                      &offset_est, &count_est);

    /*
     * If we have a known LIMIT, and don't have an unknown OFFSET, we can
     * estimate the effects of using a bounded sort.
     */
    if (count_est > 0 && offset_est >= 0)
        limit_tuples = (double) count_est + (double) offset_est;
}
```

其次，如果 SPJ 路径返回的结果本身就符合 Order By 子句要求的顺序，那么就可以免掉一

次排序的时间，因此可以根据 Order By 来生成一个 PathKeys（PathKeys 含义请参照第 6 章中的说明），用来表示 Order By 想要什么样顺序输入，这样在基于 SPJ 操作生成路径的时候可以优先考虑符合 Order By 的路径。同理，Group By 子句如果选择基于 Sort 的方法生成分组，如果 SPJ 操作生成的路径满足 Group By 子句想要的 Sort 顺序，那么也能节省一个 Sort 的代价，因此也会生成一个 PathKeys 来提示 SPJ 路径生成时最好生成一个符合 PathKeys 的路径。按照 PostgreSQL 的语法规则，如果一个语句中既有 Order By，又有 Group By，那么使用 Group By 子句产生的 PathKeys 对下层进行提升。

```
//standard_qp_callback 函数，基于 Group By 子句的 PathKeys 在第 1 顺位
if (root->group_pathkeys)
   root->query_pathkeys = root->group_pathkeys;
//基于窗口函数子句的 PathKeys 在第 2 顺位
else if (root->window_pathkeys)
   root->query_pathkeys = root->window_pathkeys;
//基于 Order By 子句的 PathKeys 和去重操作产生的 PathKeys 在第 3 顺位
//顺位相同，PathKeys 长的优先
else if (list_length(root->distinct_pathkeys) >
       list_length(root->sort_pathkeys))
   root->query_pathkeys = root->distinct_pathkeys;
else if (root->sort_pathkeys)
   root->query_pathkeys = root->sort_pathkeys;
else
   root->query_pathkeys = NIL;
```

3.10 小结

逻辑重写优化的大的基调是减少查询的层次、消除外连接操作、减少冗余操作。本章比较重要的是子查询（包括子连接）的提升、外连接的消除、谓词规范，这些工作在现阶段还不能明显地看出有什么意义，不过随着查询优化分析的深入它们的作用就会显现出来。

逻辑优化的重写规则相对还是比较简单的，有兴趣的读者可以使用第 2 章中介绍的查询树打印函数在优化前后打印查询树，分析等价变换的内容，争取做到对查询树的内存形式烂熟于心。

在逻辑重写优化之后，查询树中已经没有了右外连接，这对后面的逻辑分解优化非常重要，因为它可以少处理一种情况，这样极大地增强了代码的可读性。

4 第4章 逻辑分解优化

逻辑分解优化是逻辑优化的一部分，从这里开始和物理优化产生了紧密的联系。例如在进行逻辑重写优化的过程中，查询树中的连接树（Query->jointree）是通过 RangeTblEntry 组织起来的，但在逻辑分解优化的过程中，所有的 RangeTblEntry 将建立一个对应的 RelOptInfo 结构体，这是因为查询树（Query）中的 RangeTblEntry 结构体在物理优化时已经不适用了，必须使用适用于物理优化的 RelOptInfo 结构体来代替它。RangeTblEntry 结构体对应的是查询语句中的范围表，它更多起到对一个表进行描述的作用，属于逻辑层面，它的内部没有提供和物理代价及物理路径相关的成员变量，而 RelOptInfo 结构体在设计的时候则更多地考虑了生成物理连接路径及计算路径代价的需要，属于物理层面。

在查询树中约束条件就是一个表达式，这时的表达式是一个“裸”的表达式，就是说它只保存表达式本身所需的内容，而在逻辑分解阶段则将这些裸表达式用 RestrictInfo 结构体来进行封装，这样就可以扩展表达式的内容，比如在 RestrictInfo 结构体中还记录了约束条件在物理优化过程中需要的变量。在查询树（Query）中，约束条件（表达式）存放的位置就是它原始的语法位置，在逻辑分解的过程中，会对这些约束条件尝试下推，RestrictInfo 结构体的出现也是为了在下推的时候能够更好地和 RelOptInfo 结构体结合。

在逻辑分解阶段会基于等价类进行推理，生成一些隐含的约束条件，例如在满足一定条件的情况下，约束条件 A=B 和约束条件 B=C 能够推导出新的约束条件 A=C，例如约束条件 A=B 和 SORT(B)能够推导出 A 和 B 同样有序，再例如约束条件 A=B 和约束条件 B=5 能够推导出新的约束条件 A=5，基于这种推理在物理优化阶段能够生成更丰富的连接路径。

由于关系数据库语法中引入了外连接和半连接，很多关系代数中的经典理论不再适用，例如在约束条件下推的过程中，由于外连接的引入导致一些连接条件被延迟下推（delay）。谓词下推、连接顺序交换、基于等价类的推理是查询优化的难点，也是重点，如果没有透彻理解逻辑分解优化，那么物理优化的部分理解起来就会更加困难。

4.1 创建 RelOptInfo

在查询树中，将基表信息以 RangeTblEntry 的形式存放在 Query->rtable 链表中，在查询优化的后期，尤其是物理优化的阶段，RangeTblEntry 保存的信息已经无法满足代价计算的需要，因为针对每个基表都需要生成扫描路径（Scan），多个基表之间还会产生连接（Join）路径，并且需要计算这些路径的代价，因此需要一个新的结构体 RelOptInfo 来替换原来的 RangeTblEntry。

4.1.1 RelOptInfo 结构体

在查询优化的过程中，我们首先面对的是 FROM 子句中的表，通常称之为范围表（RangeTable），它可能是一个常规意义上的表，也可能是一个子查询，或者还可能是一个查询结果的组织为表状的函数（例如 TableFunction），这些表处于查询执行计划树的叶子节点，是产生最终查询结果的基础，我们称之为基表，这些基表可以用 RelOptInfo 结构体表示，它的 RelOptInfo->reloptkind 是 RELOPT_BASEREL。基表之间可以进行连接操作，连接操作产生的“中间”结果也可以用 RelOptInfo 结构体来表示，它对应的 RelOptInfo->reloptkind 是 RELOPT_JOINREL。另外还有 RELOPT_OTHER_MEMBER_REL 类型的 RelOptInfo 用来表示继承表的子表或者 UNION 操作的子查询等。由于 RelOptInfo 结构体集多种“功能”于一身，因此它的体积也比较庞大，我们分成多个部分来介绍它。

第一部分是 RelOptInfo 结构体中的公共变量，无论是 RELOPT_BASEREL 类型还是 RELOPT_JOINREL 类型或其他类型，都会用到这些成员变量。

```
//不同类型的 RelOptInfo，对应 RELOPT_BASEREL/RELOPT_JOINREL 等类型
RelOptKind  reloptkind;
```

```
//对于基表类型（RELOPT_BASEREL），relids 存储的是单个基表的 rtindex
//对于连接类型（RELOPT_JOINREL），relids 存储的是参与连接操作的基表 rtindex 的集合
Relids  relids;              /* set of base relids (rangetable indexes) */

//估计 RelOptInfo 对应的结果集有多少元组（行）
double  rows;                /* estimated number of result tuples */

//对查询的物理路径做预检（add_path_precheck 函数）的时候，
//是否考虑预检启动代价（startup-cost）
bool     consider_startup; //非参数化路径使用
bool     consider_param_startup; //参数化路径使用

//是否可以考虑并行查询
bool     consider_parallel;/* consider parallel paths? */

//RelOptInfo 的查询结果对应的投影列
struct PathTarget *reltarget;   /* list of Vars/Exprs, cost, width */

//记录查询路径（Path）信息
List   *pathlist;       //所有“值的”记录的路径
List   *ppilist;        // ParamPathInfo 结构体链表，参数化路径的参数
List   *partial_pathlist; //并行路径链表
struct Path *cheapest_startup_path; //启动代价最低的路径
struct Path *cheapest_total_path; //整体代价最低的路径
struct Path *cheapest_unique_path; //唯一化路径，和 SEMI-JOIN 相关
List   *cheapest_parameterized_paths; //代价最低的参数化路径列表，
                        //包括所有的参数化路径，另外，cheapest_total_path 即使
                        //不是参数化路径，也会在这里面保存一份

//记录 Lateral 变量，在 Lateral 语法支持的部分有详细介绍
Relids        direct_lateral_relids; //语句中明确指出的 Lateral 变量
Relids        lateral_relids; //由 direct_lateral_relids 进行传递闭包，推理产生表之间的依赖关系
```

这里需要介绍一下 consider_startup 变量和 consider_param_startup 变量的作用。PostgreSQL 数据库记录了两种类型的代价，分别是启动代价和整体代价，每条物理路径都会记录启动代价和整体代价，通常代价的计算是在路径创建之后进行的，但是 PostgreSQL 数据库为了提高查询优化的性能，在一些连接路径创建之前，会计算一个初级的代价用来对这个连接路径进行“预检”，如果发现这条物理路径没什么优势，那么就不创建了，这样就能节省创建路径的消耗，consider_startup 变量和 consider_param_startup 变量用来标识在“预检”时是否比较启动代价。

启动代价通常在含有 LIMIT 子句的时候才比较重要（可参考第 6 章中关于代价的介绍），也就是说通常在 tuple_fraction>0 的时候 consider_startup 才有用。

consider_param_startup 变量的行为和 consider_startup 类似，但它主要针对的是参数化路径的“预检”，而且是在 SemiJoin 和 AntiJoin 两种连接中才起作用（参照 set_base_rel_consider_ startup 函数），因为参数化路径目前只能是 NestloopJoin 连接操作的内表路径，通常来说它的启动代价的意义不大，但是对于 SemiJoin 和 AntiJoin 这两种类型的连接操作，由于每次对内表只要匹配到一条记录就可以了，这时启动代价就有意义了。

第二部分是基表（RELOPT_BASEREL）类型必用的变量。

```
Index          relid;  //基表的 rtindex
Oid            reltablespace;     //基表的表空间
RTEKind        rtekind; //基表的可能类型 RTE_RELATION,RTE_SUBQUERY,RTE_FUNCTION

//min_attr 和 max_attr 是表中的首列和末列的编号，对于 RTE_RELATION 而言，首列是
//（FirstLowInvalidHeapAttributeNumber+1），末列则是表的真实的最后一列的编号
//对于 RTE_SUBQUERY、RTE_FUNCTION 这样的 RelOptInfo，它的首列默认是 0，末列则是
//这个 RelOptInfo 结果集的投影列的个数
AttrNumber     min_attr;     /* smallest attrno of rel (often <0) */
AttrNumber     max_attr;     /* largest attrno of rel */

// attr_needed 是一个数组，数组长度是 (max_arrt - min_arrt)
//数组中的而每个 Relids 都代表它所对应的列被哪些表引用
Relids      *attr_needed;   /* array indexed [min_attr .. max_attr] */

// attr_widths 是一个数组，数组长度是 (max_arrt - min_arrt)
//每个数组元素代表了这一列的宽度
int32       *attr_widths;   /* array indexed [min_attr .. max_attr] */

//Lateral 语义
List   *lateral_vars;  /* LATERAL Vars and PHVs referenced by rel */
Relids  lateral_referencers;     /* rels that reference me laterally */

//这个表所对应的索引列表，为每个索引生成一个 IndexOptInfo
List   *indexlist;      /* list of IndexOptInfo */

//扩展多列统计信息
List   *statlist;       /* list of StatisticExtInfo */

//获得表的页面数、元组数、可见页面的比例
```

```
BlockNumber pages;          /* size estimates derived from pg_class */
double      tuples;
double      allvisfrac;

//如果是子查询，查询优化模块会向下递归遍历，子查询生成的子执行计划使用 subroot
PlannerInfo *subroot;       /* if subquery */

//子查询的参数
List   *subplan_params; /* if subquery */

//并行度
int         rel_parallel_workers;  /* wanted number of parallel workers */
```

RelOptInfo 还记录了它自己如果作为内表的时候，是否具有 UNIQUE 特性。

```
// unique_for_rels 表示当前的 RelOptInfo 对 unique_for_rels 列表中的 Relids 对应的表
//具有 UNIQUE 特性，non_unique_for_rels 表示当前的 RelOptInfo 对 non_unique_for_rels
//没有 UNIQUE 特性
List   *unique_for_rels;
List   *non_unique_for_rels;
```

另外还记录了一些扫描路径（Scan）和连接路径（Join）有用的信息。

```
List   *baserestrictinfo;  //基表上的过滤条件
QualCost baserestrictcost; //和 baserestrictinfo 一一对应，每个表达式的执行代价
Index  baserestrict_min_security;    //安全相关

//连接条件，条件中的引用了当前基表（也可能是连接过滤条件）
List   *joininfo;

//当前 RelOptInfo 和其他 RelOptInfo 可能在等价类中存在等值连接条件
bool     has_eclass_joins; /* T means joininfo is incomplete */
```

4.1.2　IndexOptInfo 结构体

每个 IndexOptInfo 结构体代表一个索引，如果一个表有多个索引，那么就有多个 IndexOptInfo 结构体存储在 RelOptInfo->indexlist 中，这些索引信息主要用于生成索引扫描路径、计算索引扫描代价、获得索引扫描结果是否具有有序性等。例如，如果查询计划采用了 B-tree 索引扫描，因为 B-tree 索引是有序的，因此可以推理出索引扫描的结果也可以是有序的，这样就给查询优化提供了优化的空间，假如索引扫描的上层路径是 MergeJoin 路径，MergeJoin 需要对索引扫描的结果排序，但如果 MergeJoin 发现索引扫描结果本身就有序，就可以省掉去排序的代价。

```
typedef struct IndexOptInfo
{
    NodeTag       type;

    Oid           indexoid;          //索引对应的 OID
    Oid           reltablespace;     //索引对应的表空间（TABLESPACE）
    RelOptInfo *rel;                 //索引所在的基表结构体指针

    //索引的磁盘页面数、元组数、树高
    BlockNumber pages;
    double        tuples;
    int           tree_height;

    //索引元信息，ncolumns 是索引键值的数量，
    //indexkeys 是具体的每个键值
    //从 indexcollations 到 canreturn 之间的成员变量是每个键值的属性数组
    int           ncolumns;      //索引键值数
    int         *indexkeys;      //索引具体的键值
    Oid         *indexcollations;
    Oid         *opfamily;       //操作符族
    Oid         *opcintype;      //操作符类
    Oid    *sortopfamily;        //如果索引是有序的，对应 B 树的操作符族
    bool   *reverse_sort;        //键值是升序还是降序
    bool   *nulls_first;         //NULL 值在索引中的位置
    bool   *canreturn;           //索引能否直接返回索引项
    Oid      relam;              //索引的访问方法，不同类型的索引访问方法不同

    List   *indexprs;            //表达式索引中的表达式
    List   *indpred;             //索引中的谓词列表
    List   *indextlist;          //索引键值的 targetlist 形式（TargetEntry 结构体链表）

    //通常是索引所在基表的 baserestrictinfo，但如果索引包含谓词（indpred != NULL），
    //那么要把和索引谓词重复的约束条件删除掉
    List   *indrestrictinfo;

    bool      predOK;            //indpred 中的谓词能否满足 Query 查询树的需要
    bool      unique;            //UNIQUE 索引
    bool      immediate;         //是否是延迟检查的索引
    bool      hypothetical;      //是否是 hypothetical 索引，目前 GIN 类型的索引有此特性
    ......
} IndexOptInfo;
```

索引从种类上可以分为唯一索引、主键索引、局部索引和表达式索引。唯一索引和主键索引在 PostgreSQL 数据库中本质上都是唯一索引，它们通过 IndexOptInfo->unique 变量来表示唯一性，对于可延迟的唯一性约束需要通过 IndexOptInfo->immediate 来表示，如果 IndexOptInfo->immediate == false，那么就代表这个唯一性约束是可延迟的，也就是说唯一性的检查延迟到事务结束的时候，而不是在写入数据（例如 INSERT）行的时候立刻进行检查。

局部索引是指带有约束条件的索引，约束条件存储在 IndexOptInfo->indexpred 中，如果一个查询中的约束条件包含于局部索引的约束条件，那么 IndexOptInfo->predOK 可以设置为 true，这样查询优化器就可以方便地决定是否可以采用局部索引来生成扫描路径，例如：

```
postgres=# CREATE INDEX SCORE_DEGREE_PRED_IDX ON SCORE(degree) WHERE degree > 60;
CREATE INDEX
postgres=# EXPLAIN SELECT * FROM SCORE WHERE degree > 60;
                              QUERY PLAN
-------------------------------------------------------------------------------------
 Index Scan using score_degree_pred_idx on score  (cost=0.13..8.14 rows=1 width=12)
(1 row)
```

表达式索引则是指在索引的键值上存在表达式，它可以是针对一个投影列进行表达式求值，也可以是针对几个投影列的统一的投影列求值，表达式被统一存储在 IndexOptInfo->indexprs 中，只有在查询语句中出现相同的表达式时，表达式索引才可能会被查询优化器采用，例如：

```
postgres=# CREATE INDEX SCORE_DEGREE_EXPR_IDX ON SCORE((degree < 90));
CREATE INDEX
postgres=# EXPLAIN SELECT * FROM SCORE WHERE degree < 90;
QUERY PLAN
-------------------------------------------------------------------------------------
Index Scan using score_degree_expr_idx on score  (cost=0.13..8.15 rows=1 width=12)
Index Cond: ((degree < 90) = true)
Filter: (degree < 90)
(3 rows)
```

从索引形式上分，PostgreSQL 数据库把索引分成了多种种类，例如 Btree 索引、Hash 索引等，针对不同的索引 PostgreSQL 数据库提供了不同的访问方法（Access Method），访问方法保存在 PG_AM 系统表中，其中的 amhandler 是访问方法的初始化方法，Btree 索引的访问方法在 bthandler 函数中实现，Hash 索引的访问方法在 hashhandler 函数中实现，这些 XXXhandler 函数初始化 IndexAmRoutine 结构体，IndexAmRoutine 结构体中的内容包括针对索引的操作方法（共 20 个函数）和该类型的索引所具有的一些属性。在初始化 IndexOptInfo 结构体时，IndexAmRoutine 结构体中的一个方法（代价估算方法）和一些属性也会被复制进来，方便给查

询优化器提供参考。

4.1.3 创建 RelOptInfo

在 PlannerInfo 中有两个数组，分别是 simple_rte_array 和 simple_rel_array，这两个数组分别负责记录 RangeTblEntry 和 RelOptInfo，它们是一一对应的关系。

在 setup_simple_rel_arrays 函数中，会将 Query->rtable 中的 RangeTblEntry 按照顺序提取出来，记录到 simple_rte_array 中。

在 add_base_rels_to_query 函数中，会将 Query->jointree 中的 RangeTblRef 提取出来，并且按照 simple_rte_array 中记录的顺序，分别为每个 RangeTblRef 创建 RelOptInfo，实际上也就是为每个 RangeTblEntry 创建对应的 RelOptInfo。

我们知道在 Query->jointree 中有三种类型的 Node 节点，分别是 FromExpr、JoinExpr 和 RangeTblRef，而 FromExpr 和 JoinExpr 的叶子节点也一定是 RangeTblRef 结构体，因此对 Query->jointree 进行深度遍历，直到发现 RangeTblRef 节点，就创建对应的 RelOptInfo 结构体，并且将 RelOptInfo 放到对应的 Query->simple_rel_array 数组中，如图 4-1 所示。

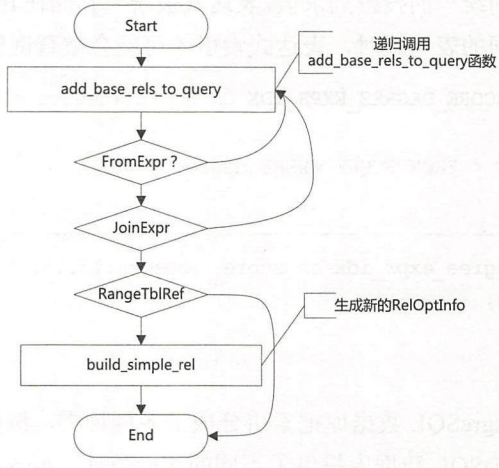


图 4-1 创建 RelOptInfo 流程图

创建 RelOptInfo 的具体过程是在 build_simple_rel 函数中实现的，这里创建的是基表（包含继承表的子表）对应的 RelOptInfo 结构体，如图 4-2 所示，基表的主要类型是 RELOPT_BASEREL 和 RELOPT_OTHER_MEMBER_REL 两种类型，基表又可以根据 RangeTblEntry-> rtekind 引申出来多种更多的类型，最常见的有普通表、子查询、VALUES 子查询、函数型基表等，如图 4-2

所示由下到上是查询实体从逻辑层面向物理层面转换的类型对照。

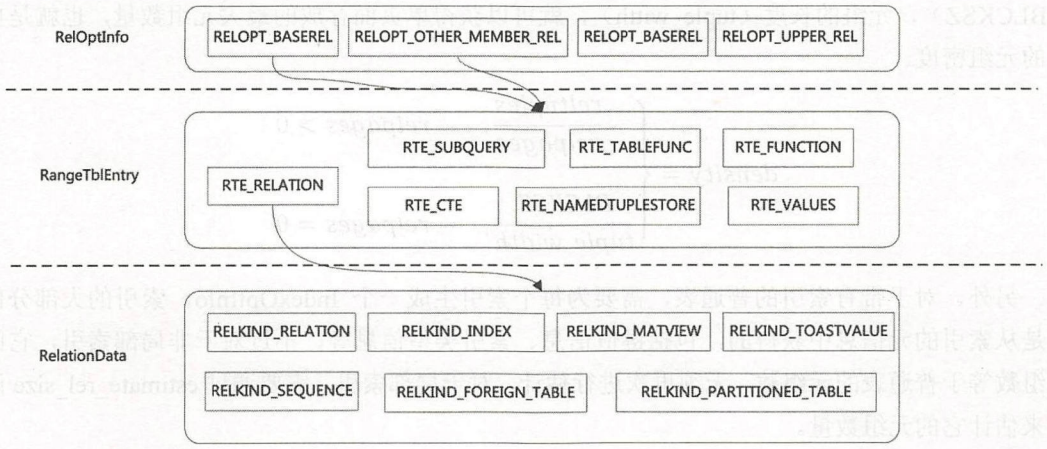


图 4-2　基表类型层次图

RelOptInfo 中的大部分变量在创建时还不能填充，例如物理路径（Path）在目前这个阶段还没有开始生成，可以先设置物理路径相关的成员变量为 NULL（默认值）。

普通表是最常用的表，它可以通过统计信息、索引信息来丰富 RelOptInfo 结构体，但是对于子查询、VALUES 类型子查询等不存在统计信息，因此它们在创建对应的 RelOptInfo 的时候和普通表不同，普通表通过 get_relation_info 函数来填充 RelOptInfo，而其他类型的基表目前只填充了 min_arrt、max_atr、attr_needed 和 attr_widths。

对于普通表的信息获取函数 get_relation_info，需要关注的有两个地方，一个是估计普通表的规模（pages、tuples），另一个是针对普通表的索引创建 IndexOptInfo 来标示索引。

普通表的规模估计是在 estimate_rel_size 函数中进行的，对于普通表的 pages（块数）数量，可以直接通过 RelationGetNumberOfBlocks 获得，获取的方式是（底层的文件大小/块大小，即 FileSize/BlockSize），但是如果要获得精确的元组数量就需要比较高的代价，因此元组数量采用的是估计的方法。在统计信息模块，对每个普通表进行统计之后，都会在 SYS_CLASS 系统表记录这个表当前的页面数和元组数的估计值，假设表上的元组数和页面数是线性增长的，那么单页面上元组的数量（元组密度）也就不会改变。我们可以通过 SYS_CLASS 里面记录的元组数和页面数来获得元组的密度，然后调用 RelationGetNumberOfBlocks 获得当前页面数，通过元组的页面密度乘以当前页面数就可以估计出当前的元组数量。

对于做过 TRUNCATE 操作的表，它在 SYS_CLASS 中记录的页面数是 0，这时可以假设元

非卖品，仅供非商业用途或交流学习使用

组在页面内是“满”的，也就是说页面上所有的空间都用于存放元组，只要我们知道页面大小（BLCKSZ），元组的长度（tuple_witth），就可以获得单页面存放的最大元组数量，也就是页面的元组密度。

$$density = \begin{cases} \dfrac{reltuples}{relpages}, & relpages > 0 \\ \dfrac{BLCKSZ}{tuple\_width}, & relpages = 0 \end{cases}$$

另外，对于带有索引的普通表，需要为每个索引生成一个 IndexOptInfo，索引的大部分信息是从索引的元信息中获得的，包括键值信息、索引类型信息等，不过对于非局部索引，它的元组数等于普通表的元组数，无须再次进行估计，对于局部索引，需要通过 estimate_rel_size 函数来估计它的元组数量。

4.2 初识等价类

等价类的处理是在约束条件下推的后期引入的，但在介绍约束条件下推之前，我们需要对等价类有一个初步的认识，因此这里先对等价类做一个说明性的介绍，在 SQL 语句中，经常会有 A=B 这样的约束条件，它的操作符是等值操作符，我们将这种等值约束条件称为“等价条件”，而基于多个等价条件进行推理而获得的等价的属性的集合就是“等价类”。

假如我们在 SQL 语句中发现一个约束条件为 A=B，那么通过这个约束条件而产生连接结果中 A 和 B 一定是相等的，如果查询结果是按照 A 进行排序的，那么就可以得知查询结果也一定是按照 B 排序的。例如约束条件 STUDENT.sno = SCORE.sno ORDER BY SCORE.sno，它的查询结果中的每一个元组都符合 STUDENT.sno = SCORE.sno，因此 STUDENT.sno 和 SCORE.sno 就构成一个等价类，虽然 Order by 子句中显式指定的是按照 SCORE.sno 进行排序，但是对于等价类中的成员而言，只要其中的一个是有序的，那么在查询结果中它们就全部都是有序的。

从示例的查询计划可以看出，查询计划依照等价类推理产生了新的排序条件，不是参照 SQL 语句中显式指定的 SCORE.sno 进行排序，而是参照 STUDENT.sno 进行排序的，由于 SCORE.sno 和 STUDENT.sno 在同一个等价类中，参照其中的任何一个作为排序依据，查询结果都是完全等价的。

```
postgres=# explain select * from student,score where student.sno=score.sno order by score.sno;
                              QUERY PLAN
-----------------------------------------------------------------------------------------
```


非卖品，仅供非商业用途或交流学习使用

```
Sort  (cost=59.41..59.58 rows=71 width=23)
  Sort Key: student.sno //等价类
  ->  Hash Join  (cost=1.16..57.22 rows=71 width=23)
        Hash Cond: (score.sno = student.sno)
        ->  Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
        ->  Hash  (cost=1.07..1.07 rows=7 width=11)
              ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
(7 rows)
```

假如有两个约束条件分别是 A=B 和 B=C，根据等值传递的特性很容易推理出 A=C，这时候就可以说{A,B,C}构成一个等价类，因此我们可以隐式地构建出一个 A = C 的约束条件条件，如果在生成物理路径（Path）的过程中，由于隐式的约束条件 A = C 的存在，就可能“多”一条连接路径，或许就多了一个更好的选择。

例如对于 SQL 语句 SELECT * FROM STUDENT, SCORE, COURSE WHERE COURSE.cno = STUDENT.sno AND STUDENT.sno=SCORE.sno，按照 SQL 语句中的语义它的 COURSE 表和 SCORE 表之间是没有约束条件的，在生成物理路径阶段，如果尝试对这两个表（可以参考第 7 章中的内容）进行连接，由于它们之间没有约束条件就只能产生基于卡氏积的连接路径，但是如果基于等价类进行推理可以发现能够产生一个新的 COURSE.cno = SCORE.sno 约束条件，这个约束条件能将 COURSE 表和 SCORE 表之间的连接结果“缩小”，从而也就会降低路径的代价。从下面的查询计划可以看出，确实产生了一个新的 COURSE.cno = SOCRE.sno 的约束条件。

```
postgres=# EXPLAIN SELECT * FROM STUDENT,SCORE,COURSE WHERE COURSE.cno= STUDENT.sno AND
STUDENT.sno=SCORE.sno;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Hash Join  (cost=30.35..68.53 rows=71 width=69)
   Hash Cond: (score.sno = course.cno) //等价类
   ->  Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
   ->  Hash  (cost=30.27..30.27 rows=7 width=57)
     ->  Nested Loop  (cost=0.15..30.27 rows=7 width=57)
       ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
       ->  Index Scan using course_pkey on course  (cost=0.15..4.17 rows=1 width=46)
             Index Cond: (cno = student.sno)
(8 rows)
```

再假如有两个约束条件 A=B 和 B=5，我们就能隐式的构建出 A=5，对于 A=5 这样的单属性（只涉及一个表）约束条件，或许能够下推到基表上，这样就可以在对表进行扫描的时候把没用的元组过滤掉，从而提高执行的效率。

非卖品，仅供非商业用途或交流学习使用

例如 SQL 语句 SELECT * FROM STUDENT, SCORE WHERE STUDENT.sno=5 AND STUDENT.sno=SCORE.sno，本来只能在连接结果产生之后使用 STUDENT.sno = SCORE.sno 对连接结果进行过滤，但是如果推理出 SCORE.sno = 5 这样一个过滤条件，那么就能把这个过滤的操作下推到对 SCORE 表的扫描上。如下面的示例所示，的确产生了新的约束条件 SCORE.sno = 5。

```
postgres=# SELECT * FROM STUDENT, SCORE WHERE STUDENT.sno = 5 AND STUDENT.sno = SCORE.sno;
                        QUERY PLAN
-------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..36.69 rows=10 width=23)
   ->  Seq Scan on student  (cost=0.00..1.09 rows=1 width=11)
         Filter: (sno = 5)  //原始约束条件：STUDEN.sno = 5
   ->  Seq Scan on score  (cost=0.00..35.50 rows=10 width=12)
         Filter: (sno = 5)  //等价类推理条件：SCORE.sno = 5
(5 rows)
```

这种基于等价类的推理虽然能够帮助查询优化器产生更多的物理路径，但是同样需要注意，在引入了外连接（或半连接、反连接）之后情况就会变得复杂，例如在外连接中的 A=B 虽然是等值的连接条件，但这时我们不能草率地认为 A=B，外连接的查询结果对连接操作的 Nullable-side 可能会补 NULL 值，这时候 A 和 B 就不是相等的。对于大多数外连接的情况我们都无法生成等价类，但是其中也会有一些特例，我们可以在上面做做文章。

从下面示例的查询计划可以看出，在对 SCORE 表进扫描时增加了 SCORE.sno = 1 作为过滤条件，我们仔细分析这个语句发现，增加 SCORE.sno = 1 并不会改变原来语句的执行结果，原因如下：首先 STUDENT.sno = 1 作为过滤条件（注意：STUDENT.sno = 1 能够进行谓词下推，下推到 STUDENT 表上），能够在表扫描的过程中把 STUDENT 中所有 STUDENT.sno != 1 的元组过滤掉，因此扫描结果中的元组一定符合 STUDENT.sno = 1，那么用这个扫描结果和 SCORE 表做约束条件为 STUDENT.sno = SCORE.sno 的左连接，SCORE.sno != 1 的元组就不可能连接上，因此增加新的过滤条件 SCORE.sno = 1 不会对连接结果产生影响。

```
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN SCORE ON STUDENT.sno = SCORE.sno WHERE
STUDENT.sno=1;
                        QUERY PLAN
-----------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..2.11 rows=1 width=23)
   Join Filter: (student.sno = score.sno)
   ->  Seq Scan on student  (cost=0.00..1.09 rows=1 width=11)
         Filter: (sno = 1)
   ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
```


非卖品，仅供非商业用途或交流学习使用

```
      Filter: (sno = 1)
(6 rows)
```

我们通过示例介绍了等价类适用的多种情况，对等价类有了初步的认识，现在稍稍深入一下，介绍等价类的数据结构——EquivalenceClass 的结构体，它的定义如下：

```
typedef struct EquivalenceClass
{
    NodeTag       type;
    List   *ec_opfamilies; //操作符族
    Oid     ec_collation;  //排序校正的变量
    List   *ec_members;    //等价类的成员链表（EquivalenceMember 结构体链表）
    List   *ec_sources;    //产生这个等价类的约束条件（RestrictInfo 结构体链表）
    List   *ec_derives;    //基于推理产生出来的约束条件（RestrictInfo 结构体链表）
    Relids   ec_relids;    //等价类成员中涉及的所有的表的 rtindex 集合
    bool     ec_has_const; //等价类成员中是否有常量
    bool     ec_has_volatile;       //等价类成员中是含有易失性函数
    bool     ec_below_outer_join;   //是否是外连接之下的等价类
    bool     ec_broken;             //推理产生的约束条件应用失败，设置此标记
    Index     ec_ sortref;          //和 Order by 相关的等价类
    Index    ec_min_security;       //安全相关
    Index     ec_max_security;      //安全相关
    struct EquivalenceClass *ec_merged; //当前等价类已经合并到另一个等价类中
} EquivalenceClass;
```

等价类又是由等价类成员结构体构成的，等价类成员结构体的定义如下：

```
typedef struct EquivalenceMember
{
    NodeTag       type;
    Expr   *em_expr;       //等价类成员的表达式，就是“=”两边的变量，多数情况下是 Var 或常量
    Relids   em_relids;    //em_expr 中引用的表的 rtindex 集合
   //等价类成员的源约束条件是否涉及下层外连接的 Nullable-side
    Relids   em_nullable_relids;
    bool     em_is_const;  //当前等价类成员是不是常量
    bool     em_is_child;  基于继承表的推理，这种等价类成员作用不大
    Oid      em_datatype;  //表达式的数据类型
} EquivalenceMember;
```

例如对于 SQL 语句 SELECT * FROM STUDENT LEFT JOIN (SCORE INNER JOIN COURSE ON SCORE.cno = COURSE.cno AND COURSE.cno = 5) ON STUDENT.sno = SCORE.SNO WHERE STUDENT.sno = 1，最终会产生 3 个等价类，这 3 个等价类会保存在 PlannerInfo->

非卖品，仅供非商业用途或交流学习使用

eq_classes 中，其结构如图 4-3 所示。

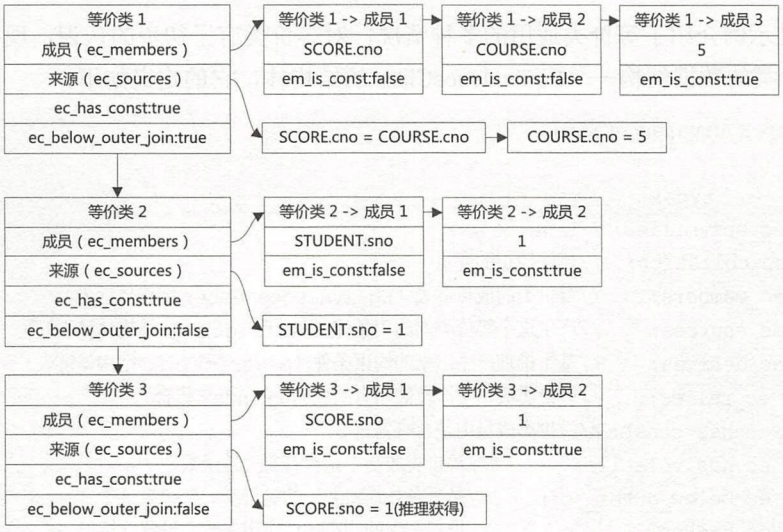


图 4-3　等价类内存结构图

4.3　谓词下推

RelOptInfo 结构体生成之后是保存在数组中的，也就是说这些基表已经从树状结构（Query->jointree）被拉平成线性结构（PlannerInfo->simple_rel_array），随之而来的问题是基表之间的逻辑关系无法在 PlannerInfo->simple_rel_array 数组上得到体现，例如在查询树中记录了有哪些基表做了什么类型的连接（记录在 FromExpr、JoinExpr 中），各个基表之间有哪些约束条件（包括连接条件和过滤条件），这些使用树状结构比较容易表现，而对于数组这种“扁平化”的结构则不容易表现，因此，逻辑分解优化要做的工作一方面是将约束条件下推到 RelOptInfo，另一方面就是要将基表之间的连接类型记录起来。

4.3.1　连接条件的下推

我们将约束条件细分成了连接条件和过滤条件，它们的作用略有不同，连接条件强调的是连接操作的计算过程，由于外连接会对没有连接上的元组补 NULL 值，所以对于 A=B 这样的连接条件产生出来的连接结果不一定满足 A=B，有可能产生出来的是 A 和 NULL。而过滤条件强调的是对查询结果的过滤作用，主要是对信息的筛选。

非卖品，仅供非商业用途或交流学习使用

对连接条件而言，如果只有内连接，连接条件下推的过程就变得简单了，连接条件可以直接下推到自己所涉及的基表上。例如下面的示例中的约束条件 student.sno == 1，细分的话它是一个连接条件，但这个连接条件在执行计划中又被下推到了 STUDENT 表的扫描路径上，变成一个过滤条件，这样就减少了 STUDENT 表扫描的结果数量，就能降低连接操作的代价，如图 4-4 所示是对示例的图形化描述。

```
postgres=# EXPLAIN SELECT * FROM STUDENT INNER JOIN SCORE ON STUDENT.sno = 1;
                        QUERY PLAN
-------------------------------------------------------------------
 Nested Loop  (cost=0.00..51.89 rows=2040 width=23)
   ->  Seq Scan on student  (cost=0.00..1.09 rows=1 width=11)
         Filter: (sno = 1)
   ->  Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
(4 rows)
```

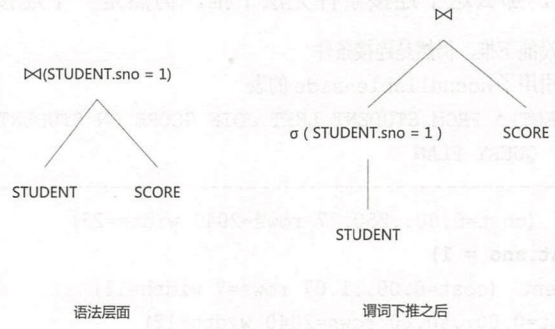


图 4-4　连接条件下推示意图

需要注意的是，这个连接条件下推之后不再是一个连接条件，变成了一个过滤条件（Filter），过滤条件主要起到信息筛选的作用，因此这个过滤条件对 STUDENT 表进行了筛选，我们可以把它引申成一个结论。

结论 1：连接条件下推之后会变成过滤条件，过滤条件下推之后仍然是过滤条件。

但是在引入了外连接和（反）半连接之后，情况就变得复杂起来，不同的连接类型给连接操作符两端的表赋予了不同的属性：Nonnullable-side 和 Nullable-side。通常来说在外连接中 Nonnullable-side 的表没匹配上连接条件的元组也会显示出来，并且在表 Nullable-side 补 NULL 值。以左外连接为例，处于 LHS 的表是 Nonnullable-side，因为它不符合连接条件的元组也会被投影出来，而处于 RHS 的表是 Nullable-side，Nonnullable-side 不符合连接条件的元组投影时，对应 Nullable-side 的部分投影的是 NULL 值，表 4-1 中总结了各种连接类型对应的 Nullable-side

非卖品，仅供非商业用途或交流学习使用

和 Nonnullable-size（对于 Nullable-side 和 Nonnullable-side 在不同的位置它们的含义可能略有不同）。

表 4-1　Nonnullable-side和Nullable-side与逻辑操作符的关系

连接类型	Nonnullable-side	Nullable-side
A Left Join B	A	B
A Right Join B	B	A
A Full JoinB	A、B	A、B
A Semi Join B	A、B	无
A Anti JoinB	A	B

由于外连接的 Nullable-side 一端可能要补 NULL 值，连接条件的下推就会受到阻碍，如果连接条件引用了 Nullable-side 的表，连接条件是可以下推变成过滤条件的，如果连接条件引用了 Nonnullable-side 的表，那么这个连接条件无法下推，仍然是一个连接条件，如下面的示例。

```
--例句是左外连接，连接条件没能下推，仍然是连接条件
--连接条件未能下推是因为其引用了 Nonnullable-side 的表
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN SCORE ON STUDENT.sno = 1;
                        QUERY PLAN
------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..250.77 rows=2040 width=23)
   Join Filter: (student.sno = 1)
   -> Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
   -> Materialize  (cost=0.00..40.60 rows=2040 width=12)
        -> Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
(5 rows)

--例句是右外连接，连接条件可以下推，变成了过滤条件
--连接条件能够下推的原因是其引用的是 Nullable-side 的表
postgres=# EXPLAIN SELECT * FROM STUDENT RIGHT JOIN SCORE ON STUDENT.sno = 1;
                        QUERY PLAN
------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..56.99 rows=2040 width=23)
   -> Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
   -> Materialize  (cost=0.00..1.09 rows=1 width=11)
        -> Seq Scan on student  (cost=0.00..1.09 rows=1 width=11)
             Filter: (sno = 1)
(5 rows)
```

那么如果“强制”下推引用了 Nonnullable-side 的连接条件会有什么后果呢？答案是会导致

非卖品，仅供非商业用途或交流学习使用

外连接的语义发生改变，也就产生了“非等价变换”，例如我们对 SQL 语句：

```
SELECT * FROM STUDENT LEFT JOIN SCORE ON STUDENT.sno = 1;
```

把连接条件 STUDENT.sno = 1 强制下推，那么转换成的语句是：

```
SELECT * FROM (SELECT * FROM STUDENT WHERE STUDENT.sno = 1) AS st LEFT JOIN SCORE ON TRUE;
```

按照左连接的语义，Nonnullable-side 表中的所有元组都应该被投影出来，而“强制”下推连接条件后，对 Nonnullable-side 表中的元组进行了过滤，导致可能有部分元组无法显示出来，因此这种“强制”下推是错误的。

```
--STUDENT 表中的数据
postgres=# SELECT sno, sname, ssex FROM STUDENT;
 sno  |  sname    | ssex
------+----------+------
   1  | zhangsan |    1
   2  | lisi     |    1
(2 rows)

--SCORE 表中的数据
postgres=# SELECT sno, cno, degree FROM SCORE;
 sno  | cno | degree
------+-----+--------
   1  |   1 |    36
(1 row)

--连接条件不能下推的查询结果
postgres=# SELECT * FROM STUDENT LEFT JOIN SCORE ON STUDENT.sno = 1;
 sno  |  sname    | ssex | sno | cno | degree
------+----------+------+-----+-----+--------
   1  | zhangsan |    1 |   1 |   1 |    36
   2  | lisi     |    1 |     |     |
(2 rows)

--强制下推之后产生的查询结果
postgres=# SELECT * FROM (SELECT * FROM STUDENT WHERE STUDENT.sno = 1) AS st LEFT JOIN SCORE
ON TRUE;
 sno  |  sname    | ssex | sno | cno | degree
------+----------+------+-----+-----+--------
   1  | zhangsan |    1 |   1 |   1 |    36
(1 row)
```


非卖品，仅供非商业用途或交流学习使用

由此，我们可以引申出一个新的结论。

结论 2：如果连接条件引用了 Nonnullable-side 的表，那么连接条件不能下推，如果连接条件只引用了 Nullable-side 的表，那么连接条件可以下推。

目前示例中的连接条件只考虑了“Var = Const”这种情况，而对于“Var = Var”“Var Op Var = Var”“Func(Var) = Var”等情况都没有验证，我们只需要把握一个原则，就是如果连接条件中引用了 Nonnullable-side 的表，那么这个连接条件就不能下推，用户可以尝试构建用例来对这个结论进行验证。

目前只考虑了两个表的情况，《道德经》中有“一生二，二生三，三生万物”的说法，我们尝试用 3 个表来代表普遍的情况，看一个包含 3 个表连接的 SQL 语句：

```
SELECT * FROM STUDENT LEFT JOIN (SCORE LEFT JOIN COURSE ON TRUE) ON STUDENT.sno = 1;
```

按照示例语句的语义，它的查询树应该如图 4-5 所示。

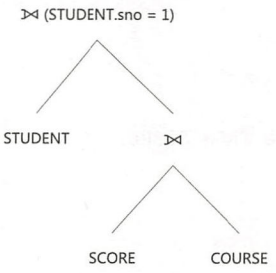


图 4-5　多个表连接的约束条件下推示例

如果我们把其中的（SCORE LEFT JOIN COURSE ON TRUE）看成一个整体，那么顶层左连接的 Nullable-side 就是（SCORE LEFT JOIN COURSE ON TRUE），顶层左连接的 Nonnullable-side 就是 STUDENT 表，因为 STUDENT.sno = 1 引用了 STUDENT 表中的列（也就是引用了 Nonnullable-side 的表的列），因此 STUDENT.sno = 1 不能被下推。

```
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN (SCORE LEFT JOIN COURSE ON TRUE) ON
STUDENT.sno = 1;
                          QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..393080.12 rows=2244000 width=69)
   Join Filter: (student.sno = 1) //连接条件没有被下推
   -> Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
   -> Nested Loop Left Join  (cost=0.00..28104.15 rows=2244000 width=58)
        -> Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
```

```
        ->  Materialize  (cost=0.00..26.50 rows=1100 width=46)
              ->  Seq Scan on course  (cost=0.00..21.00 rows=1100 width=46)
(7 rows)
```

我们再来看一个语句：

```
SELECT * FROM STUDENT LEFT JOIN (SCORE LEFT JOIN COURSE ON TRUE) ON SCORE.cno = COURSE.cno;
```

连接条件所引用的表都在（SCORE LEFT JOIN COURSE ON TRUE）中，我们如果把（SCORE LEFT JOIN COURSE ON TRUE）看成一个整体，那么它就处于顶层左连接的 Nullable-side，也就是说连接条件是能下降的，结合结论 1 和结论 2，它们下降之后首先变成这样：

```
SELECT * FROM STUDENT LEFT JOIN (SELECT * FROM SCORE LEFT JOIN COURSE ON TRUE WHERE SCORE.cno
= COURSE.sno) ON TRUE;
```

从查询计划可以看出，连接条件下推之后和下推之前它们的执行计划是相同的。

```
--显然，两个执行计划是相同的
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN (SCORE LEFT JOIN COURSE ON TRUE) ON SCORE.cno
= COURSE.cno;
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=34.75..275.56 rows=14280 width=69)
   ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
   ->  Materialize  (cost=34.75..101.09 rows=2040 width=58)
         ->  Hash Join  (cost=34.75..90.89 rows=2040 width=58)
               --已经被下推，但要注意，变成过滤条件后可以将
               --外连接消除，目前 SCORE 和 COURSE 两个表做的是内连接的 Hash Join
               Hash Cond: (score.cno = course.cno)
               ->  Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
               ->  Hash  (cost=21.00..21.00 rows=1100 width=46)
                     ->  Seq Scan on course  (cost=0.00..21.00 rows=1100 width=46)
(8 rows)

postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN (SELECT * FROM SCORE LEFT JOIN COURSE ON
TRUE WHERE SCORE.cno = COURSE.cno) sc
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=34.75..275.56 rows=14280 width=69)
   ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
   ->  Materialize  (cost=34.75..101.09 rows=2040 width=58)
         ->  Hash Join  (cost=34.75..90.89 rows=2040 width=58)
               Hash Cond: (score.cno = course.cno)
```

```
          -> Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
          -> Hash  (cost=21.00..21.00 rows=1100 width=46)
               -> Seq Scan on course  (cost=0.00..21.00 rows=1100 width=46)
(8 rows)
```

4.3.2 过滤条件的下推

我们再给出一些例句，这些语句的连接条件也只涉及了 Nullable-side 的表，因此也是能够下推的。

例 1：

下推前：SELECT * FROM STUDENT LEFT JOIN (SCORE LEFT JOIN COURSE ON TRUE) ON SCORE.cno = 1;

下推后：SELECT * FROM STUDENT LEFT JOIN (SELECT * FROM SCORE LEFT JOIN COURSE ON TRUE WHERH SCORE.cno = 1) ON TRUE;

例 2：

下推前：SELECT * FROM STUDENT LEFT JOIN (SCORE LEFT JOIN COURSE ON TRUE) ON COURSE.cno = 1;

下推后：SELECT * FROM STUDENT LEFT JOIN (SELECT * FROM SCORE LEFT JOIN COURSE ON TRUE WHERE COURSE.cno = 1) ON TRUE;

上面例句的下推是没有问题的，但如果我们进一步追问：连接条件已经变成了过滤条件，那么这个子查询中的过滤条件能够继续下推吗？

连接条件下推到子查询中之后变成了过滤条件，我们把例 1 和例 2 中“下推后”语句的子查询提取出来，单独看这两个子查询的执行计划：

```
--过滤条件 SCORE.cno = 1 被下推到 SCORE 表上，执行计划里仍然是左连接
postg=# EXPLAIN SELECT * FROM SCORE LEFT JOIN COURSE ON TRUE WHERE SCORE.cno = 1;
                        QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..196.75 rows=11000 width=58)
   -> Seq Scan on score  (cost=0.00..35.50 rows=10 width=12)
        Filter: (cno = 1)
   -> Materialize  (cost=0.00..26.50 rows=1100 width=46)
        -> Seq Scan on course  (cost=0.00..21.00 rows=1100 width=46)
```

```
(5 rows)

--过滤条件 COURSE.cno = 1 也被下推到 COURSE 表上，需要注意这时候外连接被消除了
--很显然，过滤条件 COURSE.cno = 1 是“严格”的
postg=# EXPLAIN SELECT * FROM SCORE LEFT JOIN COURSE ON TRUE WHERE COURSE.cno = 1;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Nested Loop  (cost=0.15..58.97 rows=2040 width=58)
   ->  Index Scan using course_pkey on course  (cost=0.15..8.17 rows=1 width=46)
         Index Cond: (cno = 1)
   ->  Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
(4 rows)
```

通过例子我们可以看出，如果过滤条件只引用了 Nonnullable-side 的表，那么这个过滤条件能够下推到 Nonnullable-side 的表上，如果过滤条件引用了 Nullable-side 的表且过滤条件是严格的（关于“严格”的解释可以参考外连接消除章节），那么会导致外连接消除变成内连接。在内连接的情况下，过滤条件显然也能下推到对应的表上，那么如果过滤条件引用了 Nullable-side 的表且过滤条件是不严格的，情况会怎样呢？如下面的示例所示，这种情况过滤条件是不能下推的。

```
// COURSE.cno IS NULL 是不严格的
postgres=# EXPLAIN SELECT * FROM SCORE LEFT JOIN COURSE ON TRUE WHERE COURSE.cno IS NULL;
                          QUERY PLAN
 Nested Loop Left Join  (cost=0.00..28104.15 rows=11220 width=58)
   Filter: (course.cno IS NULL) //过滤条件没有下推
   ->  Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
   ->  Materialize  (cost=0.00..26.50 rows=1100 width=46)
         ->  Seq Scan on course  (cost=0.00..21.00 rows=1100 width=46)
(5 rows)
```

由此，我们得到一个新的结论。

结论 3：如果过滤条件只引用了 Nonnullable-side 的表，那么这个过滤条件能够下推到表上，如果过滤条件引用了 Nullable-side 的表且过滤条件是严格的，那么会导致外连接消除，外连接消除之后变成内连接，过滤条件也是能下推的。

4.3.3　连接顺序

各位读者是否注意到，我们在构建约束条件下推的示例时，有意避开了一种情况：

（A leftjoin B）　leftjoin　C

请看下面示例中的 SQL 语句，如果从连接条件下推的角度来看，COURSE.cno = SCORE.cno 是无法下推的，因为它引用了 Nonnullable-side 的表（这里把 STUDENT LEFT JOIN COURSE ON TRUE 看作一个整体），但是通过查看下面 SQL 语句的执行计划会发现执行计划改变了表之间的连接顺序，由原来的(STUDENT LEFT JOIN COURSE ON TRUE) LEFT JOIN SCORE ON COURSE.cno = SCORE.cno 变化成了 STUDENT LEFT JOIN (COURSE LEFT JOIN SCORE ON COURSE.cno = SCORE.cno) ON TRUE，使得连接条件同样地下降了一层。

```
-- ( A LEFT JOIN B ) LEFT JOIN C
postgres=# EXPLAIN SELECT * FROM (STUDENT LEFT JOIN COURSE ON TRUE) LEFT JOIN SCORE ON COURSE.cno
= SCORE.cno;
                          QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=1.02..126.23 rows=7700 width=69)
   ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
   ->  Materialize  (cost=1.02..31.66 rows=1100 width=58)
         ->  Hash Left Join  (cost=1.02..26.16 rows=1100 width=58)
               Hash Cond: (course.cno = score.cno)
               ->  Seq Scan on course  (cost=0.00..21.00 rows=1100 width=46)
               ->  Hash  (cost=1.01..1.01 rows=1 width=12)
                     ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
(8 rows)
```

查询优化器在尝试生成连接路径的时候会尝试交换基表之间的连接顺序，目的是生成更多的候选路径，也就有更大的概率选出更优的路径，但是连接顺序并不能随意交换。

对于内连接而言，因为它符合关系代数的经典理论，因此可以任意交换顺序，最坏的情况无非是所有的表做卡氏积，但是如果增加了外连接（Outer Join，包含 Left Outer Join、Right Outer Join、Full Outer Join）、半连接（Semi Join）、反半连接（Anti Join）这 3 种逻辑连接类型之后，连接顺序就要受到严格的限制。

由于在外连接消除的逻辑优化过程中同时消除了右外连接（如果发现右外连接则交换连接顺序，变成左外连接），PostgreSQL 数据库在逻辑分解优化阶段只有全连接（Full Outer Join）和左连接（Left Outer Join），再由于 PostgreSQL 数据库对全连接的情况不允许交换连接顺序，因此 PostgreSQL 数据库在注释文档中给出了基于左连接的顺序交换的等式，其中 A、B、C 为参与连接的基表，Pab 代表引用了 A 表和 B 表上的列的谓词（连接条件）。

等式 1.1：(A leftjoin B on (Pab)) innerjoin C on (Pac)
= (A innerjoin C on (Pac)) leftjoin B on (Pab)

等式 1.2：(A leftjoin B on (Pab)) leftjoin C on (Pac)

= (A leftjoin C on (Pac)) leftjoin B on (Pab)

等式 1.3：(A leftjoin B on (Pab)) leftjoin C on (Pbc)

= A leftjoin (B leftjoin C on (Pbc)) on (Pab)

&Pbc must be strict

通过分析发现，(STUDENT LEFT JOIN COURSE ON TRUE) LEFT JOIN SCORE ON COURSE.cno = SCORE.cno 符合的是等式 1.3，下面再给出等式 1.1 和等式 1.2 的示例。

```
// 等式1.1
postgres=# explain SELECT * FROM (STUDENT LEFT JOIN COURSE ON TRUE) INNER JOIN SCORE ON
STUDENT.sno = SCORE.sno;
                          QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=1.16..1057.22 rows=78100 width=69)
   ->  Hash Join  (cost=1.16..57.22 rows=71 width=23)
         Hash Cond: (score.sno = student.sno)
         ->  Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
         ->  Hash  (cost=1.07..1.07 rows=7 width=11)
               ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
   ->  Materialize  (cost=0.00..26.50 rows=1100 width=46)
         ->  Seq Scan on course  (cost=0.00..21.00 rows=1100 width=46)
(8 rows)

// 等式1.2
postgres=# explain SELECT * FROM (STUDENT LEFT JOIN COURSE ON TRUE) LEFT JOIN SCORE ON
STUDENT.sno = SCORE.sno;
                          QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=1.16..1057.22 rows=78100 width=69)
   ->  Hash Right Join  (cost=1.16..57.22 rows=71 width=23)
         Hash Cond: (score.sno = student.sno)
         ->  Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
         ->  Hash  (cost=1.07..1.07 rows=7 width=11)
               ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
   ->  Materialize  (cost=0.00..26.50 rows=1100 width=46)
         ->  Seq Scan on course  (cost=0.00..21.00 rows=1100 width=46)
(8 rows)
```

对于半连接（Semi Join），PostgreSQL 数据库的注释文档中指出：

```
   SEMI joins work a little bit differently. A semijoin can be reassociated into or out of
the lefthand side of another semijoin, left join, or antijoin, but not into or out of the
righthand side. Likewise, an inner join, left join, or antijoin can be reassociated into
or out of the lefthand side of a semijoin, but not into or out of the righthand side.
```

由此我们引申出关于半连接的等式 2，其中 innerjoin/leftjoin/semijoin/antijoin 任选其一。

等式 2：(A semijoin B ON Pab) innerjoin/leftjoin/semijoin/antijoin C ON Pac
= (A innerjoin/leftjoin/semijoin/antijoin C ON Pac) semijoin B ON Pab

我们以(A semijoinB ON Pab) leftjoin C ON Pac 为例来查看一下执行计划。

```
// (A semijoin B ON Pab) leftjoin C ON Pac
postgres=# EXPLAIN SELECT * FROM (SELECT * FROM SCORE WHERE SCORE.sno > ANY (SELECT sno FROM
STUDENT)) sc LEFT JOIN COURSE ON sc.cno = COURSE.cno;
                        QUERY PLAN
-------------------------------------------------------------------------
 Hash Left Join  (cost=34.75..235.46 rows=680 width=58)
   Hash Cond: (score.cno = course.cno)
   -> Nested Loop Semi Join  (cost=0.00..192.14 rows=680 width=12)
        Join Filter: (score.sno > student.sno)
        -> Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
        -> Materialize  (cost=0.00..1.11 rows=7 width=4)
             -> Seq Scan on student  (cost=0.00..1.07 rows=7 width=4)
   -> Hash  (cost=21.00..21.00 rows=1100 width=46)
        -> Seq Scan on course  (cost=0.00..21.00 rows=1100 width=46)
(9 rows)
```

从示例可以看出，查询计划中并没有按照我们预想的进行顺序交换，这里没有交换的原因是根据代价计算，不交换的执行计划代价比较低，我们反向地看一下，如果(A semijoin B ON Pab) leftjoin C ON Pac 等价于(A leftjoin C ON Pac) semijoin B ON Pab，那么(A leftjoin C ON Pac) semijoin B ON Pab 产生的执行计划应该与上面示例中的执行计划相同。

```
postgres=# EXPLAIN SELECT * FROM (SCORE LEFT JOIN COURSE ON SCORE.cno = COURSE.cno) sc WHERE
sc.sno > ANY(SELECT sno FROM STUDENT);
                        QUERY PLAN
-------------------------------------------------------------------------
 Hash Left Join  (cost=34.75..235.46 rows=680 width=58)
   Hash Cond: (score.cno = course.cno)
   -> Nested Loop Semi Join  (cost=0.00..192.14 rows=680 width=12)
        Join Filter: (score.sno > student.sno)
```

```
      ->  Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
      ->  Materialize  (cost=0.00..1.11 rows=7 width=4)
            ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=4)
  ->  Hash  (cost=21.00..21.00 rows=1100 width=46)
      ->  Seq Scan on course  (cost=0.00..21.00 rows=1100 width=46)
(9 rows)
```

通过示例可以看出查询优化器对(A leftjoin C ON Pac) semijoin B ON Pab 产生的执行计划做了顺序交换，转变成了(A semijoin B ON Pab) leftjoin C ON Pac 的形式。

对于反半连接也具有同样的性质。

等式 3：(A antijoin B ON Pab) innerjoin/leftjoin/semijoin/antijoin C ON Pac
= (A innerjoin/leftjoin/semijoin/antijoin C ON Pac) antijoin B ON Pab

我们也看一下反半连接的一个例子。

```
-- 直接用(A leftjoin C ON Pac) antijoin B ON Pab 验证
postgres=# EXPLAIN SELECT * FROM (SCORE LEFT JOIN COURSE ON SCORE.cno = COURSE.cno) sc WHERE
not exists (SELECT sno FROM STUDENT WHERE sc.sno=STUDENT.sno);
                        QUERY PLAN
------------------------------------------------------------------------
 Hash Left Join  (cost=35.91..95.88 rows=1020 width=58)
   Hash Cond: (score.cno = course.cno)
   ->  Hash Anti Join  (cost=1.16..48.26 rows=1020 width=12)
         Hash Cond: (score.sno = student.sno)
         ->  Seq Scan on score  (cost=0.00..30.40 rows=2040 width=12)
         ->  Hash  (cost=1.07..1.07 rows=7 width=4)
               ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=4)
   ->  Hash  (cost=21.00..21.00 rows=1100 width=46)
         ->  Seq Scan on course  (cost=0.00..21.00 rows=1100 width=46)
(9 rows)
```

可以看出查询优化器对(A leftjoin C ON Pac) antijoin B ON Pab 产生的执行计划做了顺序交换，转变成了(A antijoin B ON Pab) leftjoin C ON Pac 的形式，转换后的等价的 SQL 语句如下：

```
SELECT * FROM (SELECT * FROM SCORE WHERE not exists (SELECT sno FROM STUDENT WHERE
SCORE.sno=STUDENT.sno)) sc LEFT JOIN COURSE ON sc.cno = COURSE.cno;
```

读者可以自己尝试查看一下执行计划是否相同。

需要注意的是，连接顺序交换等价性的形式化证明并非易事，对于 PostgreSQL 数据库的开发人员也是如此，而且即使能够证明其等价，通过清晰的代码来实现也有一定的难度，上面的

5 个等式可能并不全面，需要在分析查询优化的过程中根据不同的情况灵活地进行分析，不过我们可以基于一个大的原则。

SemiJoin 通常和 InnerJoin 具有一定的相似性，因为可以把 SemiJoin 看作（InnerJoin+内表唯一化）的情况，因此它本质上还是通过连接操作来产生结果的。

AntiJoin 通常和 LeftJoin 具有一定的相似性，因为 AntiJoin 需要的是和内表匹配不上，而 LeftJoin 恰好也把和内表匹配不上的数据显示出来，因此它们两个都天然地含有“补 NULL”的逻辑。

这种大的原则虽然不“精确”，但它是有用的，可以帮助我们在分析具体问题的时候更好地理清连接操作之间的逻辑关系。

4.3.4 deconstruct_recurse 函数

无论是在子查询提升阶段还是预处理表达式阶段我们都对 Query->jointree 进行过递归处理，在逻辑分解优化的过程中，仍然需要对 Query->jointree 进行递归处理，这次处理之后，Query->jointree 就正式“退休”了，后续的主要工作就由 Query->simple_rel_array 接手了。虽然 Query->simple_rel_array 中的 RelOptInfo 已经创建好了，但是它的信息还没有填充完，我们这次递归遍历的主要目的是给这些 RelOptInfo 分发约束条件，并且要把它们的逻辑连接关系建立好。

deconstruct_recurse 函数对 Query->jointree 的递归遍历仍然分成 3 部分，分别处理 RangeTblRef、FromExpr 和 JoinExpr，并且对 FromExpr 和 JoinExpr 做深度遍历，直到发现 RangeTblRef 叶节点，在 deconstruct_recurse 函数中有一些比较重要的变量，我们需要重点说明一下。

qualscope 变量记录了在当前连接层次之下出现的所有表的 rtindex 集合。

inner_join_rels 变量记录了在当前连接层次之下所有的内连接涉及的表的 rtindex 集合。

nullable_rels/ nonnullable_rels 两个变量出现在对 JoinExpr 进行处理的时候，它们分别记录了连接中的 Nullable-side/Nonnullable-side 表的 rtindex 集合。

below_outer_join 变量默认是 false，在递归遍历的过程中，如果遇到了外连接和（反）半连接，这个值就会进行调整。在对外连接的 Nullable-side 进行遍历时，需要递归调用 deconstruct_recurse 函数，below_outer_join 变量在这时设置为 true；在对反半连接的 RHS 端进行遍历时，同样需要递归调用 deconstruct_recurse 函数，below_outer_join 变量这时也被设置为 true。

postponed_qual_list 变量的引入是由于增加了 Lateral 语义之后，如果 Lateral 子查询被提升，那么在产生的子连接树里的约束条件可以引用上层连接的表的变量，这个约束条件无法在当前的子连接树上分发（distribute_qual_to_rels 函数）给 RelOptInfo，需要记录下来返回给上层连接，由上层连接负责分发给 RelOptInfo，因此把这类需要推迟分发的约束条件记录到 postponed_qual_list 链表中。

例如对于语句 SELECT * FROM STUDENT LEFT JOIN LATERAL (SELECT * FROM SCORE WHERE STUDENT.sno = SCORE.sno) sc ON TRUE，子查询提升之后的 jointree 如图 4-6 所示，STUDENT.sno = SCORE.sno 所在的 FromExpr 没有 STUDENT 表，因此这个约束条件需要被推迟分发，因此把这个约束条件返回给上层，直到约束条件引用的表在上层被收集全，才会进行约束条件分发。

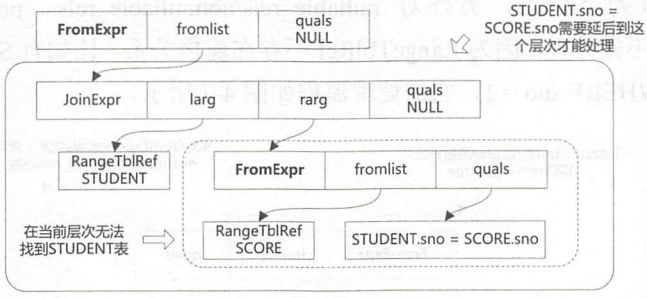


图 4-6　约束条件的延迟处理

postponed_qual_list 的收集过程在 distribute_qual_to_rels 函数中实现，它会比较当前约束条件中的 rtindex 集合和当前层次的表的 rtindex 集合，如果当前约束条件上的 rtindex 集合不是当前层次的表的 rtindex 集合的子集，那么就将当前的约束条件记录到 postponed_qual_list 变量。

```
//relids 是从约束条件中提取的 rtindex 集合
//qualscope 是当前层次的表的 rtindex 集合
//bms_is_subset 是子集判断函数
if (!bms_is_subset(relids, qualscope))
{
    PostponedQual *pq = (PostponedQual *) palloc(sizeof(PostponedQual));
    pq->qual = clause;
    pq->relids = relids;
    *postponed_qual_list = lappend(*postponed_qual_list, pq);//收集到postponed_qual_list
    return;
}
```

deconstruct_recurse 函数中有 postponed_qual_list 变量和 child_postponed_quals 变量，这两个

变量都是需要推迟分发的约束条件，postponed_qual_list 变量是当前层次返回给上层的约束条件链表，child_postponed_quals 是下层返回给当前层次的约束条件链表。

joinlist 变量用来返回拉平的连接树，但它的长度受到 from_collapse_limit 变量和 join_collapse_limit 变量的限制。from_collapse_limit 变量限制 FromExpr->fromlist 的节点的个数，如果超过 from_collapse_limit 的值，就不再展开。join_collapse_limit 变量的功能和 from_collapse_limit 变量类似，如果出现了多个连续的连接且超出了 join_collapse_limit 的限制，那么就不再展开。但需要注意一个例外情况，对于全连接（Full Join）的两个节点是不展开的。

然后我们介绍 deconstruct_recurse 函数处理 RangeTblRef、FromExpr 和 JoinExpr 的流程。

对于 RangeTblRef 节点，设置 qualscope 为当前 RangeTblRef 对应的 rtindex，设置 inner_join_rels 的值为 NULL，另外对 nullable_rels/nonnullable_rels、postponed_qual_list、below_outer_join 都不做处理，因为 RangeTblRef 不存在连接关系，比如有 SQL 语句 SELECT * FROM STUDENT WHERE sno = 1，它的处理流程如图 4-7 所示。

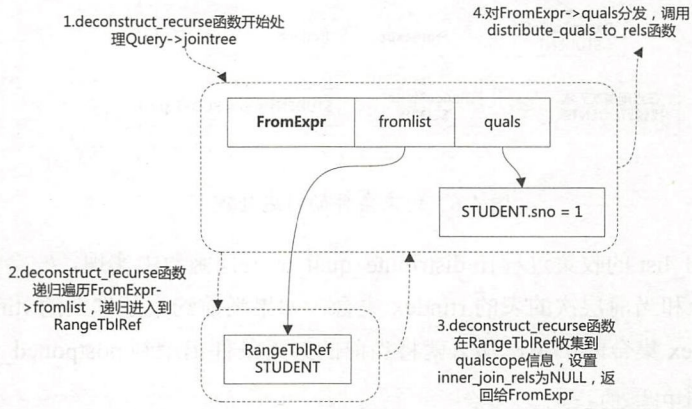


图 4-7　分解连接树的流程

对于 FromExpr 节点，首先需要对 FromExpr->fromlist 进行遍历，每处理 FromExpr->fromlist 的中的一个节点，都会返回一个 sub_qualscope，我们要把这些 sub_qualscope 统统合并到 qualscope 中 。

```
foreach(l, f->fromlist)//遍历 fromlist
{
    ......
    //递归调用 deconstruct_recurse 函数
    //返回的值有 sub_joinlist、sub_qualscope、child_postponed_quals
```

```
    sub_joinlist = deconstruct_recurse(root, lfirst(l),
                                       below_outer_join,
                                       &sub_qualscope,
                                       inner_join_rels,
                                       &child_postponed_quals);
//收集 sub_qualscope 到 qualscope
*qualscope = bms_add_members(*qualscope, sub_qualscope);
   ……
}
```

deconstruct_recurse 函数递归处理 FromExpr->fromlist 的每个节点时，处理结果返回的 sub_joinlist 都会整理到 joinlist 变量中，只是长度受到 from_collapse_limit 变量的限制。

FromExpr->fromlist 中的这些节点默认都是内连接（Inner Join），因此如果 list_length (f->fromlist) > 1，那么 inner_join_rels 就应该和 qualscope 相同。

在递归遍历 FromExpr->fromlist 的过程中，它的子节点中可能存在需要推到上层的约束条件，这些约束条件返回到 child_postponed_quals 中。在 FromExpr 中将考虑对 child_postponed_quals 进行处理，因为这时候的 qualscope 已经变成了整个 FromExpr->fromlist 中涉及的所有节点。如果约束条件引用的表是在当前 qualscope 的子集，那么这个条件就可以通过 distribute_qual_to_rels 函数进行分发了。如果约束条件中引用的表仍然不是 qualscope 的子集，那么就把这个约束条件放到 postponed_qual_list 中返回给上层去处理。这里可以更加明确地看出 child_postponed_quals 是下层返回给当前层次的约束条件链表，而 postponed_qual_list 是当前层次返回给上层的约束条件链表。child_postponed_quals 变量和 postponed_qual_list 变量的处理流程分析如下：

```
//child_postponed_quals 是下层返回来的推迟约束条件，到当前层次尝试能够分发
foreach(l, child_postponed_quals)
{
    PostponedQual *pq = (PostponedQual *) lfirst(l);

    //推迟约束条件涉及的表是不是当前层次表的子集
    if (bms_is_subset(pq->relids, *qualscope))
        distribute_qual_to_rels(……);//是子集，能够分发就进行分发
    else
        //推迟约束条件需要继续被推迟，就再返回给更上层
        *postponed_qual_list = lappend(*postponed_qual_list, pq);
}
```

对于 JoinExpr 的处理需要递归遍历 JoinExpr->larg 和 JoinExpr->rarg，其中 qualscope 就是 larg

和 rarg 中的表对应的 rtindex 集合，inner_join_rels 就是 larg 和 rarg 中的内连接涉及的表的 rtindex 集合。child_postponed_quals 变量返回的是处理 larg 和 rarg 的过程中发现的需要被推迟的约束条件。另外 below_outer_join 变量、nullable_rels/ nonnullable_rels 变量需要根据连接类型的不同进行不同的处理。

```
switch (j->jointype)
{
   case JOIN_INNER:
      leftjoinlist = deconstruct_recurse(……); //below_outer_join默认
      rightjoinlist = deconstruct_recurse(……); //below_outer_join默认
      *qualscope = bms_union(leftids, rightids);
      *inner_join_rels = *qualscope;
      nonnullable_rels = NULL;
      nullable_rels = NULL;
      break;
   case JOIN_LEFT:
   case JOIN_ANTI:
      leftjoinlist = deconstruct_recurse(……); //below_outer_join默认
      rightjoinlist = deconstruct_recurse(……); //below_outer_join = true
      *qualscope = bms_union(leftids, rightids);
      *inner_join_rels = bms_union(left_inners, right_inners);
      nonnullable_rels = leftids;
      nullable_rels = rightids;
      break;
   case JOIN_SEMI:
      leftjoinlist = deconstruct_recurse(……); //below_outer_join默认
      rightjoinlist = deconstruct_recurse(……); //below_outer_join = true
      *qualscope = bms_union(leftids, rightids);
      *inner_join_rels = bms_union(left_inners, right_inners);
      nonnullable_rels = NULL;
      nullable_rels = NULL;
      break;
   case JOIN_FULL:
       leftjoinlist = deconstruct_recurse(……); //below_outer_join = true
      rightjoinlist = deconstruct_recurse(……); //below_outer_join = true
      *qualscope = bms_union(leftids, rightids);
      *inner_join_rels = bms_union(left_inners, right_inners);
      //注意这里对Full Join做了特殊处理，它的nonnullable_rels和nullable_rels相同
      //因为nonnullable_rels在distribute_qual_to_rels函数中用来表示外连接（Outer Join），
      //如果设置为NULL，distribute_qual_to_rels函数就会认为当前连接是内连接（Inner Join）
      nonnullable_rels = *qualscope;
```

```
            nullable_rels = *qualscope;
            break;
        default:
            ……
            break;
    }
```

deconstruct_recurse 函数中的变量对 distribute_qual_to_rels 函数及 make_outerjoininfo 函数非常重要，约束条件是否能下推、连接顺序能否交换都是靠这些变量控制的。我们通过一个例子来加深一下对这些变量的认识，假如有如下 SQL 语句：

```
SELECT * FROM STUDENT LEFT JOIN (TEACHER INNER JOIN (SELECT * FROM COURSE WHERE NOT EXISTS
(SELECT * FROM SCORE WHERE COURSE.cno = SCORE.cno)) sc ON TRUE) ON TRUE;
```

在进行 deconstruct_jointree 之前 Query->jointree 的树结构如图 4-8 所示。

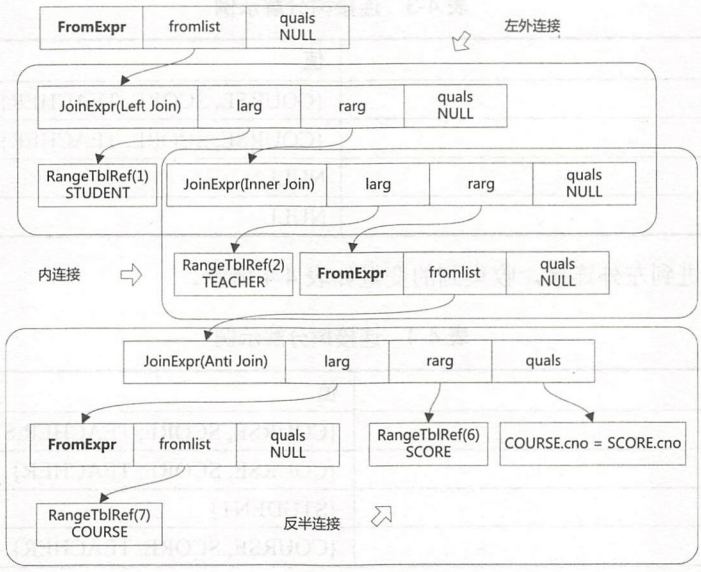


图 4-8　连接树分解前的查询树内存结构

我们从下层到上层依次看各个变量的变化情况。

反半连接处于最下层，它的左表是一个 FromExpr，在处理 RangeTblRef（COURSE）的时候，收集到一个 rtindex == 7 存放到 qualscope，inner_join_rels 设置为 NULL，然后返回给上层的 FromExpr 处理的过程，因为 fromlist 中只有一个 RangeTblRef（COURSE），因此在这个阶段 qualscope 还是只有一个表 COURSE，inner_join_rels 还是 NULL，这些都将返回给上层的反

半连接（Anti Join）。

反半连接的右表是一个 RangeTblRef，因此返回给反半连接的变量 qualscope 是 SCORE，inner_join_rels 设置为 NULL，最终反半连接收集到的变量如表 4-2 所示。

表 4-2　连接树分解示例

变量名	值
qualscope	{COURSE, SCORE}
inner_join_rels	NULL
nonnullable_rels	{COURSE}
nullable_rels	{SCORE}

再往上层递进，内连接收集到的变量如表 4-3 所示。

表 4-3　连接树分解示例

变量名	值
qualscope	{COURSE, SCORE, TEACHER}
inner_join_rels	{COURSE, SCORE, TEACHER}
nonnullable_rels	NULL
nullable_rels	NULL

在向上层递进到左外连接，收集到的变量如表 4-4 所示。

表 4-4　连接树分解示例

变量名	值
qualscope	{COURSE, SCORE, TEACHER,STUDENT}
inner_join_rels	{COURSE, SCORE, TEACHER}
nonnullable_rels	{STUDENT}
nullable_rels	{COURSE, SCORE, TEACHER}

4.3.5　make_outerjoininfo 函数

在前面章节中我们已经介绍了有关外连接的连接顺序交换的原则，并给出了能够等价交换的等式，那么如何来表示一个语句中有哪些合法的连接顺序呢？PostgreSQL 数据库定义了 SpecialJoinInfo 结构体，一方面用来记录在 deconstruct_recurse 函数中发现的外连接、（反）半连接，即记录表与表之间的逻辑连接关系，另一方面对于不能进行连接顺序交换的表也需要通过这个结构体来限制。

```
typedef struct SpecialJoinInfo
{
    NodeTag       type;
    Relids        min_lefthand;        //用来限制连接顺序，LHS 出现的最小集
    Relids        min_righthand;       //用来限制连接顺序，RHS 出现的最小集
    Relids        syn_lefthand;        //Query->jointree 中的 LHS 集合
    Relids        syn_righthand;       //Query->jointree 中的 RHS 集合
    JoinType      jointype;            //连接类型：Left Join、Full Join、Semi Join、Anti Join
    bool          lhs_strict;          //用来限制连接顺序，有些顺序交换需要保证约束条件严格
    bool          delay_upper_joins;   //用来限制连接顺序，阻止当前连接和上层交换顺序
    bool          semi_can_btree;      //连接条件中的操作符是否属于 btree 操作符族
    bool          semi_can_hash;       //连接条件中的操作符是否支持进行 hash
    List          *semi_operators;     //提取连接条件中的操作符，形成链表
    List          *semi_rhs_exprs;     //提取连接条件中的右操作数，形成链表
} SpecialJoinInfo;
```

我们先来看一下 min_lefthand/min_righthand 和 syn_lefthand/syn_righthand，min_lefthand/min_righthand 的作用是获得要形成某个连接关系，它的 LHS 和 RHS 最少需要哪些表，syn_lefthand/syn_righthand 是语法体现出来的 LHS 和 RHS 引用了哪些表，例如语句：

```
SELECT * FROM STUDENT LEFT JOIN SCORE ON STUDENT.sno = SCORE.sno LEFT JOIN COURSE ON SCORE.cno
= COURSE.cno;
```

这个 SQL 语句会形成两个 SpecialJoinInfo，第一个 SpecialJoinInfo 用来表示（STUDENT LEFT JOIN SCORE），它们的 syn_lefthand/syn_righthand 就是连接关系的 STUDENT/SCORE，它的 min_lefthand/min_righthand 和 syn_lefthand/syn_righthand 是相同的，如表 4-5 所示。

表 4-5　“最小集”的示例

成员变量	值
min_lefthand	STUDENT
min_righthand	SCORE
syn_lefthand	STUDENT
syn_righthand	SCORE
lhs_strict	true for STUDENT.sno = SCORE.sno

第二个 SpecialJoinInfo 用来表示（STUDENT LEFT JOIN SCORE LEFT JOIN COURSE），它们的 min_lefthand/min_righthand 和 syn_lefthand/syn_righthand 如表 4-6 所示，其中 min_righthand 和 syn_righthand 是相同的，但是 min_lefthand 和 syn_lefthand 不同。

表 4-6 “最小集”的示例

成员变量	值
min_lefthand	STUDENT
min_righthand	COURSE
syn_lefthand	STUDENT, SCORE
syn_righthand	COURSE
lhs_strict	true for SCORE.cno = COURSE.cno

通过观察两个 SpecialJoinInfo 的 min_lefthand/min_righthand 发现，它的 min_lefthand 都是 STUDENT 表，也就是说对这个 SQL 语句而言，STUDENT 的顺序是不能交换出去的，它必须处于左连接的 LHS，而 SCORE 和 COURSE 在 RHS 是能交换顺序的，这就符合我们介绍连接顺序交换的等式。

(A leftjoin B on (Pab)) leftjoin C on (Pbc)
= A leftjoin (B leftjoin C on (Pbc)) on (Pab)
& Pbc must be strict.

等式需要 Pbc 也就是 SCORE.cno = COURSE.cno 是严格的，这个严格的属性也记录到了 lhs_strict 成员变量中，在处理第二个 SpecialJoinInfo 的时候，会根据第一个 SpecialJoinInfo 中的 lhs_strict 来判断是否符合等式，lhs_strict 用来表示 Pbc 约束条件是否是“严格的”。

另外在 SpecialJoinInfo 结构体中还有一个 delay_upper_joins 变量，它主要用来处理一种特殊情况。

A leftjoin (B leftjoin C ON Pbc WHERE Pc) ON Pab
& Pc is not strict;

例如 SQL 语句：

```
SELECT * FROM STUDENT LEFT JOIN ( SELECT * FROM SCORE LEFT JOIN COURSE ON TRUE WHERE COURSE.cno
is NULL ) sc ON TRUE;
```

它的子查询中的 SCORE 和 COURSE 必须先做连接，也就是说子查询语句中的表 SCORE 和 COURSE 不能和上层的 STUDENT 交换连接顺序。

另外对于 SQL 语句：

```
SELECT * FROM STUDENT LEFT JOIN ( SELECT COURSE.cno FROM SCORE LEFT JOIN COURSE ON TRUE) sc
ON sc.cno IS NULL;
```

版权所有，严禁传播，违者自负法律责任！

因为 sc.cno IS NULL 能下推到下层的外连接（SELECT COURSE.cno FROM SCORE LEFT JOIN COURSE ON TRUE)之中，且下推之后会变成过滤条件，实际上和上面的语句是等价的。

在介绍了 SpecialJoinInfo 结构体之后再介绍一下 make_outerjoininfo 函数的参数情况，我们把参数的介绍总结在表 4-7 中。

表 4-7　make_outerjoininfo函数的参数说明

参数名	参数类型	描述
root	[IN] PlannerInfo *	查询优化模块的上下文信息结构体
left_rels	[IN] Relids	连接的LHS的rtindex集合
right_rels	[IN] Relids	连接的RHS的rtindex集合
inner_join_rels	[IN] Relids	连接下的所有的内连接的rtindex集合
jointype	[IN] JoinType	连接类型
clause	[IN] List *	当前连接的约束语句
返回值	Node *	SpecialJoinInfo结构体指针

make_outerjoininfo 函数首先处理了 FOR [KEY] UPDATE/SHARE 在 Nullable-side 的情况，FOR [KEY] UPDATE/SHARE 不能涉及外连接的 Nullable-side。

```
postgres=# SELECT * FROM STUDENT LEFT JOIN SCORE ON TRUE FOR UPDATE;
ERROR:  FOR UPDATE cannot be applied to the nullable side of an outer join
```

在 make_outerjoininfo 函数中才检查这种情况看似有点“晚”，这种检查通常应该在语义分析阶段来做，但是在语义分析阶段并不能获取到查询树的全部信息，例如对于含有外连接的视图（View），在语义分析阶段就无法获得视图的结构，直到做了查询重写之后，视图才以子查询的方式加入整个查询树中，因此在语义阶段无法检查这种情况。

```
-- 视图中包含外连接的情况
postgres=# CREATE VIEW SC AS SELECT * FROM STUDENT LEFT JOIN COURSE ON TRUE;
CREATE VIEW
postgres=# SELECT * FROM SC FOR UPDATE;
ERROR:  FOR UPDATE cannot be applied to the nullable side of an outer join
```

在检查了 FOR [KEY] UPDATE/SHARE 之后，就开始生成 SpecialJoinInfo 结构体，其中 syn_lefthand/syn_righthand/jointype 直接等同于参数中的 left_rels/right_rels/jointype。

对于 SpecialJoinInfo 结构体中和半连接（Semi Join）相关的变量，make_outerjoininfo 函数通过调用 compute_semijoin_info 函数实现，对于半连接而言，外表无须匹配内表的所有元组，因此可以对内表进行“唯一化”，这样就能减少内表的大小，提高匹配速度，这里通过调用

版权所有，严禁传播，违者自负法律责任！

compute_semijoin_info 函数获得了半连接的信息，在生成执行路径的时候，会考虑根据当前生成的信息生成唯一化路径（create_unique_path 函数）。

另外对于全连接的情况，我们把它的 min_lefthand/min_righthand 设置成和 syn_lefthand/syn_righthand 相同的值，也就是说全连接只能按照查询语句中语义的顺序进行连接操作，不能和其他表交换连接顺序。

lhs_strict变量的初始化主要通过find_nonnullable_rels函数来实现，在介绍reduce_outer_joins函数（外连接消除）的时候对这个函数进行了介绍，这里通过这个函数把约束条件中的“严格”的表找出来，然后将这些表和 left_rels 取交集。

```
strict_relids = find_nonnullable_rels((Node *) clause); //获得约束条件中严格的表
sjinfo->lhs_strict = bms_overlap(strict_relids, left_rels); //和当前的 LHS 的表取交集
```

然后就是 make_outerjoininfo 函数的重点工作，生成 min_lefthand/min_righthand 两个最小集。最小集的生成流程主要包含了 3 个步骤：赋予初值、遍历下层 SpecialJoinInfo 限制连接顺序、查看全局的 PlaceHolderVar 链表（root->placeholder_list），把涉及 Nullable-side 的表加到最小集。

给 min_lefthand/min_righthand 赋初值：

$$\text{min\_lefthand} = \text{clause\_relids} \cap \text{left\_rels}$$
$$\text{min\_righthand} = \{\text{clause\_relids} \cup \text{inner\_join\_rels}\} \cap \text{right\_rels}$$

其中 clause_relids 是从约束条件提取出来的所有表的 rtindex 集合，inner_join_rels、left_rels 和 right_rels 都是 make_outerjoininfo 函数的参数。

这里在 min_righthand 中加入 inner_join_rels 是因为下层的内连接 Inner Join 不能和上层的外连接交换。

A leftjoin (B innerjoin C on (Pbc)) on (Pab)
!= (A leftjoin B on (Pab)) innerjoin C on (Pbc)

在获得初值之后，就可以考虑当前层次的连接是否可以和下层的外连接、（反）半连接交换顺序。由于是递归遍历，因此当前连接的所有下层外连接都已经存放在 root->join_info_list 中，需要注意的是 root->join_info_list 中的 SpecialJoinInfo 不一定是当前层次的下层连接，例如 ((A leftjoin B) leftjoin C) leftjoin ((D leftjoin E) leftjoin F) 的连接树如图 4-9 所示。

版权所有，严禁传播，违者自负法律责任！

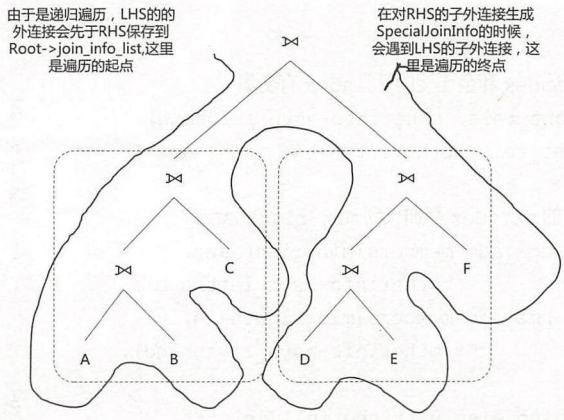


图 4-9　连接关系的生成流程

只有 root->join_info_list 中的 SpecialJoinInfo 的 Relids 和当前的 left_rels 和 right_rels 有交集的时候，才被认为是当前连接的子连接树。

在遍历 root->join_info_list 的过程中，又将其中的 SpecialJoinInfo 分成了 3 种情况，分别是 Full Join 的情况、SpecialJoinInfo 处在 LHS 的情况和 SpecialJoinInfo 处在 RHS 的情况。

对于 Full Join 的情况，如果当前层次的外连接 LHS 的表和全连接中的表有交集，那么就把全连接中的表加入 min_lefthand，如果当前层次的外连接的 RHS 和全连接中的表有交集，那么就把全连接中的表加入 min_righthand。例如，对于 A leftjoin (B fulljoin C) 的 min_lefthand 是{A}，min_righthand 是{B,C}，这样做之后 B 和 C 必须先做连接操作，然后它们的连接结果才能和 A 做连接操作。再例如对于(A fulljoin B) left join C 的 min_lefthand 是{A,B}，min_righthand 是{C}，同理，A 和 B 也必须先做全连接操作，然后它们的连接结果和 C 再做连接操作，也就是说下层的全连接不能和上层的任何类型的连接交换连接顺序。

```
if (otherinfo->jointype == JOIN_FULL)
{
   //当前连接的 LHS 的 rtindex 和全连接的 rtindex 有交集
   if (bms_overlap(left_rels, otherinfo->syn_lefthand) ||
      bms_overlap(left_rels, otherinfo->syn_righthand))
{
      //下层全连接的所有的 rtindex 都加入 min_lefthand
      min_lefthand = bms_add_members(min_lefthand,
                                     otherinfo->syn_lefthand);
      min_lefthand = bms_add_members(min_lefthand,
                                     otherinfo->syn_righthand);
```

版权所有，严禁传播，违者自负法律责任！


```
}

   //当前连接的 RHS 的 rtindex 和全连接的 rtindex 有交集
   if (bms_overlap(right_rels, otherinfo->syn_lefthand) ||
      bms_overlap(right_rels, otherinfo->syn_righthand))
{
      //下层全连接的所有的 rtindex 都加入到 min_righthand
      min_righthand = bms_add_members(min_righthand,
                                      otherinfo->syn_lefthand);
      min_righthand = bms_add_members(min_righthand,
                                      otherinfo->syn_righthand);
   }
   /* Needn't do anything else with the full join */
   continue;
}
```

如果不是全连接，那么可能的就是左连接、（反）半连接，如果它们出现在 LHS，那么我们处理这么几种情况：

- (A leftjoin B on (Pab)) leftjoin C on (Pbc) & Pbc is not strict
- (A leftjoin B on (Pab)) semijoin C on (Pbc)
- (A leftjoin B on (Pab)) antijoin C on (Pbc)

注意上面的连接条件中都含有 Pbc，也就是说连接条件中引用了下层外连接的 RHS 的表。

```
//如果下层的外连接处于当前连接的 LHS
if (bms_overlap(left_rels, otherinfo->syn_righthand))
{
if (
   //是否符合 Pbc
   bms_overlap(clause_relids, otherinfo->syn_righthand) &&
      (
         // (A leftjoin B on (Pab)) semijoin C on (Pbc)
         jointype == JOIN_SEMI || //如果是半连接（Semi Join）
         // (A leftjoin B on (Pab)) antijoin C on (Pbc)
         jointype == JOIN_ANTI || //如果是反半连接（Anti Join）
         // (A leftjoin B on (Pab)) leftjoin C on (Pbc) & Pbc is not strict
         !bms_overlap(strict_relids, otherinfo->min_righthand) //Pbc 不是严格的
      )
   )
{
   //下层外连接的所有的 rtindex 都加入 min_lefthand
```

版权所有，严禁传播，违者自负法律责任！


```
        min_lefthand = bms_add_members(min_lefthand,  otherinfo->syn_lefthand);
        min_lefthand = bms_add_members(min_lefthand,  otherinfo->syn_righthand);
    }
}
```

如果下层外连接或（反）半连接出现在当前连接的 RHS，那么需要判断的模式就会更多，我们先把它们列出来：

- A leftjoin (B leftjoin C on Pbc) ON Pac
- A leftjoin (B leftjoin C on Pbc) ON Pc
- A semijoin (B leftjoin C on Pbc) ON Pac
- A antijoin (B leftjoin C on Pbc) ON Pac
- A leftjoin (B semijoin C on Pbc) ON Pac
- A leftjoin (B antijoin C on Pbc) ON Pac
- A leftjoin (B leftjoin C on Pbc) On Pab & Pbc is not strict
- A leftjoin (B leftjoin C ON Pbc WHERE Pc) ON Pab, Pc is not strict

```
//如果下层外连接出现在当前连接的 RHS
if (
    // A leftjoin (B leftjoin C on Pbc) ON Pac
    bms_overlap(clause_relids, otherinfo->syn_righthand) ||
    // A leftjoin (B leftjoin C on Pbc) ON Pc
    !bms_overlap(clause_relids, otherinfo->min_lefthand) ||
    // A semijoin (B leftjoin C on Pbc) ON Pac
    jointype == JOIN_SEMI ||
    // A antijoin (B leftjoin C on Pbc) ON Pac
    jointype == JOIN_ANTI ||
    // A leftjoin (B semijoin C on Pbc) ON Pac
    otherinfo->jointype == JOIN_SEMI ||
    // A leftjoin (B antijoin C on Pbc) ON Pac
    otherinfo->jointype == JOIN_ANTI ||
    // A leftjoin (B leftjoin C on Pbc) On Pab & Pbc is not strict
    !otherinfo->lhs_strict ||
    // A leftjoin ( B leftjoin C ON Pbc WHERE Pc) ON Pab, Pc is not strict;
    otherinfo->delay_upper_joins
    )
{
    //下层外连接的所有的 rtindex 都加入到 min_ righthand
    min_righthand = bms_add_members(min_righthand, otherinfo->syn_lefthand);
```

版权所有，严禁传播，违者自负法律责任！


```
    min_righthand = bms_add_members(min_righthand, otherinfo->syn_righthand);
}
```

虽然看上去不等式较多，但我们在前面的章节中已经总结过等式，也就是说不符合等式的全部都不能交换顺序，不等式的情况只是等式情况的补集，这部分如果要较好地理解需要多阅读一些执行计划，分析每个执行计划不能交换的原因。

4.3.6 distribute_qual_to_rels 函数

约束条件下推（谓词下推）是非常重要的优化手段，我们在前面的章节中介绍了 PostgreSQL 数据库中关于约束条件下推（谓词下推）的规则，distribute_qual_to_rels 函数的主要功能就是参照这些原则对约束条件进行下推。

4.3.6.1 参数及变量的说明

在之前的逻辑变换优化过程中，我们已经对约束条件进行了正则化，保证了约束条件的顶层形式是合取范式，deconstruct_recurse 函数会分别针对合取范式中的每一个单独的子约束条件调用 distribute_qual_to_rels 函数进行处理。

```
foreach(l, (List *) f->quals)
{
    Node      *qual = (Node *) lfirst(l); //合取范式下的约束条件
    distribute_qual_to_rels(root, qual, //一个约束条件
                            false, below_outer_join, JOIN_INNER,
                            root->qual_security_level,
                            *qualscope, NULL, NULL, NULL,
                            postponed_qual_list);
}
```

另外，distribute_qual_to_rels 函数的参数情况总结如表 4-8 所示。

表 4-8 distribute_qual_to_rels函数的参数说明

参数名	参数类型	描述
root	[IN] PlannerInfo *	查询优化模块的上下文信息结构体
clause	[IN] Node *	约束条件，可能是单独的条件，也可能是个析取范式
is_deduced	[IN] bool	在约束条件下推的过程中会记录等价类，随后会进行等价推理，基于推理的知识可以产生新的约束条件，这些约束条件进行下推的时候is_decuced == true
below_outer_join	[IN] bool	是否处于外、反、半连接的Nullable-side

版权所有，严禁传播，违者自负法律责任！

续表

参数名	参数类型	描述
jointype	[IN] JoinType	连接类型
security_level	[IN] Index	略
qualscope	[IN] Relids	从deconstruct_recurse函数中传入的参数，在介绍deconstruct_recurse函数的时候做了介绍，该参数的值相当于syn_lefthand ∪ syn_righthand
ojscope	[IN] Relids	从deconstruct_recurse函数传入的参数，该参数的值相当于min_lefthand ∪ min_righthand
outerjoin_nonnullable	[IN] Relids	从deconstruct_recurse函数传入的参数，该参数的值，代表Nonnullable-side的rtindex集合
deduced_nullable_relids	[OUT] List **	distribute_qual_to_rels函数分发的约束条件可能是推理出来的约束条件，如果推理出这个约束条件的等价类成员原来所在的约束条件中引用了下层连接的Nullable-side的表，就通过 deduced_nullable_relids来传达
postponed_qual_list	[IN] List *	需要推迟的约束条件，假如约束条件（clause参数）引用了qualscope参数之外的表，这个约束条件无法下推成功，需要被“推迟下推”

另外在 distribute_qual_to_rels 函数中还有一些局部变量，在函数的实现过程中比较重要，这里也将它们列出来做一个说明，如表 4-9 所示。

表 4-9　distribute_qual_to_rels函数的局部变量说明

变量名	说明
is_pushed_down	true：代表过滤条件，可能是WHERE子句中的过滤条件，也可能是下推之后的连接条件 false：不能下推的连接条件
outerjoin_delayed	约束条件从当前层次下推到下层之后，都会成为过滤条件，如果下层有外连接，这个过滤条件可能能继续下推，也可能下推受到阻碍，对这种受到阻碍的情况把outerjoin_delayed变量设置为true
maybe_equivalence	这是一个粗略的筛选，在distribute_qual_to_rels函数中，会对等价的约束条件生成等价类，但是对于涉及外连接的等价约束条件，不一定能够正式生成等价类，这时候maybe_equivalence就设置为false，表示在distribute_qual_to_rels函数中无法生成等价类

版权所有，严禁传播，违者自负法律责任！

续表

变量名	说明
maybe_outer_join	maybe_equivalence变量设置为false的时候如果完全不生成等价类，就会把一些能做等价推理的情况“漏”掉，我们在介绍等价类的章节时已经介绍了在包含外连接的时候，有些情况也是能够生成等价类的，因此在 maybe_equivalence 是 false 的时候用 maybe_outer_join 再做一次区分，也就是说用，如果 maybe_equivalence == false && mybe_outer_join == true，我们会生成单成员的等价类，在reconsider_outer_join_clauses函数中再去尝试将这种单成员的等价类合并成一个正式的等价类
nullable_relids	如果约束条件引用了下层外连接的Nullable-side的表，那么就把下层外连接的Nullable-side的rtindex都记录到nullable_relids中

4.3.6.2 异常情况处理

约束条件无论如何下推，最终也越不过它所引用的表，因此约束条件所引用的表是它下推的底线，通过 pull_varnos 函数先把这些表取出来，记录到 relids 作为初值。

例如对于(A leftjoin B on Pab) leftjoin C on Pab 这样的连接关系，Pab 的下推必须以表 A 和表 B 为基础，因此 relids 的初值就是{A,B}。

另外还需要处理之前提到的推迟分发的约束条件，这些条件涉及的表（relids）已经超过了当前连接所涉及的范围（qualscope），这些连接记录到 postponed_qual_list 里，一层一层地返回给上层去，直到发现在某一层满足 relids ⊆ qualscope 为止。

还有一种情况，就是我们通过 pull_varnos 函数对 clause 进行遍历之后，没有获得 relids，这就说明这个 reldis 是一个无变量的约束条件，这种无变量的约束条件有很大的优化空间，因此 PostgreSQL 数据库对其进行了单独处理。

4.3.6.3 无变量的约束条件

无变量的约束条件的含义是该约束条件不涉及任何的表，通常是常量表达式，例如 1=1 就属于永远是 true 的常量表达式，而 1>2 则属于永远是 false 的常量表达式，这种约束条件在逻辑变换优化阶段就已经进行了求值（preprocess_expression 函数），在逻辑分解优化阶段我们见到的就是一个 bool 类型的常量值，但还有无法预先求值形如 1>random()的这种含有易失性函数的表达式，这种表达式只能在执行期间才能进行求值。

PostgreSQL 数据库将常量表达式分成了 4 类情况。

- 情况 1：外连接的无变量约束条件。

版权所有，严禁传播，违者自负法律责任！

- 情况 2：内连接的无变量约束条件、含有易失性函数。
- 情况 3：内连接的无变量约束条件、不含易失性函数、below_outer_join == true。
- 情况 4：内连接的无变量约束条件、不含易失性函数、below_outer_join == false。

```
//判定是无变量约束条件，relids 是从约束条件中提取的表
if (bms_is_empty(relids))
{
   // 情况 1：外连接的无变量约束条件
   if (ojscope)
   {
      /* clause is attached to outer join, eval it there */
      relids = bms_copy(ojscope);//原位
      /* mustn't use as gating qual, so don't mark pseudoconstant */
   }
   else
   {
      //情况 2：内连接的无变量约束条件、含有易失性函数
      relids = bms_copy(qualscope);//原位
      if (!contain_volatile_functions(clause))
      {
         //情况 3：内连接的无变量约束条件、不含易失性函数、below_outer_join == true
         pseudoconstant = true;
         root->hasPseudoConstantQuals = true;

         if (!below_outer_join)
         {
            //情况 4：内连接的无变量约束条件、不含易失性函数、below_outer_join == false
            relids = get_relids_in_jointree((Node *) root->parse->jointree, false);
            qualscope = bms_copy(relids);
         }
      }
   }
}
```

基于情况 1 可以给出如下示例，对于 SCORE LEFT JOIN COURSE ON 1 > 2 而言，由于约束条件永远是 false，因此这个连接的结果投影的是 LHS 表的全部元组，以及对 RHS 端补的 NULL 值，因此可以将对 RHS 表的扫描消除掉，从执行计划可以看出对 COURSE 表的扫描变成了 Result 节点（常量约束条件通常会生成一个 One-time Filter）。约束条件 1 > 2 在示例的执行计划中表现为 Join Filter:false，它的位置处在它原有语义的位置，因为它的最小集是 ojscope =（min_lefthand ∪ min_righthand）。

版权所有，严禁传播，违者自负法律责任！


```
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN (SCORE LEFT JOIN COURSE ON 1 > 2) ON TRUE;
                          QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..2.18 rows=7 width=69)
   ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
   ->  Materialize  (cost=0.00..1.02 rows=1 width=58)
         ->  Nested Loop Left Join  (cost=0.00..1.02 rows=1 width=58)
               Join Filter: false
               ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
               ->  Result  (cost=0.00..0.00 rows=0 width=46)
                     One-Time Filter: false
(8 rows)
```

基于情况 2 从我们给出的示例可以看出，因为约束条件含有易失性函数，因此只能在执行期才能求值，所以将其保留在原来的语法位置，是比较保险的做法。

```
postgres=# EXPLAIN SELECT * FROM STUDENT INNER JOIN (SCORE INNER JOIN COURSE ON 1 > RANDOM())
ON TRUE;
                          QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..71.71 rows=2567 width=69)
   ->  Nested Loop  (cost=0.00..38.51 rows=367 width=58)
         Join Filter: ('1'::double precision > random())
         ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
         ->  Seq Scan on course  (cost=0.00..21.00 rows=1100 width=46)
   ->  Materialize  (cost=0.00..1.11 rows=7 width=11)
         ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
(7 rows)
```

基于情况 3 从我们给出的示例可以看出，当前的 SCORE INNER JOIN COURSE ON 1 > 2 处于左连接的 Nullable-side，因此 blow_outer_join 参数的值是 true，这个约束条件会保留在它的原始的语法位置，但是会记录它的常量形式，以便查询优化器继续进行优化（注：在生成物理连接路径的阶段会检查常量约束条件）。

```
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN (SCORE INNER JOIN COURSE ON 1 > 2) ON TRUE;
                          QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..1.14 rows=7 width=69)
   ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
   ->  Result  (cost=0.00..0.00 rows=0 width=58)
         One-Time Filter: false
(4 rows)
```

基于情况 4 从我们给出的示例可以看出，由于子句中(SCORE INNER JOIN COURSE ON 1 > 2)没有结果，导致整个连接都没有结果（全是内连接），但情况 4 是把约束条件 1>2（对应执行计划中的 One-Time Filter: false）提到了最顶层。

```
postgres=# EXPLAIN SELECT * FROM STUDENT INNER JOIN (SCORE INNER JOIN COURSE ON 1 > 2) ON
TRUE;
                QUERY PLAN
------------------------------------------
 Result  (cost=0.00..0.00 rows=0 width=69)
   One-Time Filter: false
(2 rows)
```

4.3.6.4　检查约束条件下推

在前面的章节把约束条件细分成了**连接条件**和**过滤条件**，并且详细地介绍了哪些约束条件能下推，哪些约束条件不能下推。另外，还介绍过在 distribute_qual_to_rels 函数中的局部变量 is_pushed_down 可以用来区分连接条件和过滤条件，下面就来具体地分析一下这部分的实现过程。

我们将约束条件下推检查的部分分成 3 部分来介绍，第一部分是通过推理产生的约束条件，这部分是通过 is_deduced 参数来标识的。

```
//基于推理产生的约束条件
if (is_deduced)
{
    //约束条件一定是过滤条件或者能下推的连接条件
    //因为约束条件两端的变量在同一个等价类，必然是完全等价的
    is_pushed_down = true;

    //不能出现下推受到延迟的情况，如果出现下推延迟的情况，就违背了等价类的定义
    outerjoin_delayed = false;

    //推理出这个约束条件的等价类成员原来所在的约束条件中，
    //如果引用了下层连接的 Nullable-side 的表，那么这里也需要处理
    nullable_relids = deduced_nullable_relids;
    //这个约束条件是通过等价类推理产生的，因此不要再去尝试基于它生成等价类，
    //因此 maybe_equivalence 和 maybe_outer_join 都设置成 false
    maybe_equivalence = false;
    maybe_outer_join = false;
}
```

第二部分是对连接条件进行检查，如果连接条件引用了 Nonnullable-side 的表，那么这个连接条件是不能下推的。

```
//outerjoin_nonnullable 是通过参数传递进来的，它代表 Nonnullable-side 的表
else if (bms_overlap(relids, outerjoin_nonnullable))
{
    //这是一个不能下推的连接条件，is_pushed_down 设置为 false
    is_pushed_down = false;

    //约束条件的两端的变量不等价，不能用来产生等价类
    maybe_equivalence = false;

    //虽然不能产生有等价成员的等价类，
    //但可以分别给约束条件的两端的变量单独生成等价类（两个单成员等价类）
    //到 reconsider_outer_join_clauses 函数会基于单成员等价类去做基于外连接的推理
    maybe_outer_join = true;//外连接

    //收集一下这个连接条件，是否引用了下层外连接的 Nullable-side 的表
    outerjoin_delayed = check_outerjoin_delay(root,
                                    &relids,
                                    &nullable_relids,
                                    false);

    //虽然不能下推，但也不一定就必须保持在原有的语法位置，
    //表之间的连接顺序可能会产生交换，因此只要保持在它所涉及的表的最小集就可以
    Assert(ojscope);
    relids = ojscope;
    Assert(!pseudoconstant);
}
```

第三部分是对过滤条件或者能下推的连接条件进行处理。

```
//过滤条件或者能下推的连接条件
else
{
    //这些条件是能进行下推的
    is_pushed_down = true;

    //但如果下层有外连接，下推可能会被阻碍
    outerjoin_delayed = check_outerjoin_delay(root,
                                    &relids,
                                    &nullable_relids,
```


非卖品，仅供非商业用途或交流学习使用

```
                                        true);

    //如果下推被阻碍了
    if (outerjoin_delayed)
    {
        //下推被阻碍，这个约束条件可能引用了下层外连接的 Nullable-side 的表
        //那么这个等价关系就不成立了
        maybe_equivalence = false;//有下层外连接

        //如果是 IS NULL 约束条件，而且当前是 Anti Join，这个约束条件可以被消除掉
        if (check_redundant_nullability_qual(root, clause))
            return;
    }
    else
    {
        //如果下推没有被阻碍，那么就可以尝试产生基于这个约束条件的等价类
        maybe_equivalence = true;

        //outerjoin_nonnullable 如果不等于 NULL，说明这是一个外连接的约束条件
        if (outerjoin_nonnullable != NULL)
            below_outer_join = true; //是外连接
    }
    //如果能对外连接约束条件产生等价类，
    //约束条件必须引用 Nonnullable-side 的表
    //如果没有引用 Nonnullable-side 的表，就不用去尝试了
    maybe_outer_join = false;
}
```

4.3.6.5　check_outerjoin_delay 函数

check_outerjoin_delay 函数有两个作用。

- 检查约束条件下推的过程中是否会受到阻碍，并且收集约束条件中涉及的 Nullable-side 的表，也会将阻碍下推的表记录起来。
- 设置连接顺序不允许交换的标记（SpecialJoinInfo->delay_upper_joins），如果不允许交换设置为 true，这个下层的外连接就不能和上层交换连接顺序，这个标记用来处理一种特殊情况：A leftjoin (B leftjoin C ON Pbc WHERE Pc) ON Pab, Pc is not strict;。

受到阻碍的连接条件主要是不能再用来生成等价类，因为一旦引用了 Nullable-side 的表，连接结果中可能就会补充 NULL 值，导致应用约束条件后出来的结果不一致，我们分析一下源代码，看看这部分是怎么判断受到阻碍并且收集 Nullable-side 的表的。

非卖品，仅供非商业用途或交流学习使用

```
//如果引用了Nullable-side的表
//注：对于Full Join，它的LHS也是Nullable-side
if (bms_overlap(relids, sjinfo->min_righthand) ||
    (sjinfo->jointype == JOIN_FULL &&
     bms_overlap(relids, sjinfo->min_lefthand)))
{
    //是否已经包含到我们收集的表里了
    if (!bms_is_subset(sjinfo->min_lefthand, relids) ||
        !bms_is_subset(sjinfo->min_righthand, relids))
    {
        //这个外连接阻碍了约束条件下推，把它记录到relids，最终会记录到RestrictInfo结构体中
        relids = bms_add_members(relids, sjinfo->min_lefthand);
        relids = bms_add_members(relids, sjinfo->min_righthand);

        //发现被阻碍了
        outerjoin_delayed = true;
        found_some = true;
    }
    //记录外连接的Nullable-side
    nullable_relids = bms_add_members(nullable_relids,
                                      sjinfo->min_righthand);

    //如果是Full Join, LHS也记录下来
    if (sjinfo->jointype == JOIN_FULL)
        nullable_relids = bms_add_members(nullable_relids,
                                          sjinfo->min_lefthand);


//处理一种特殊情况：A leftjoin ( B leftjoin C ON Pbc WHERE Pc) ON Pab, Pc is not strict;
//详情参照make_outerjoininfo函数的介绍
    if (is_pushed_down && sjinfo->jointype != JOIN_FULL &&
        bms_overlap(relids, sjinfo->min_lefthand))
        sjinfo->delay_upper_joins = true;
}
```

4.3.6.6 生成 RestrictInfo

约束条件就是 WHERE/ON/HAVING 子句中的各个条件，在查询树中，它们以表达式（Expr）的方式存在，主要存放在 FromExpr 的 quals 链表中和 JoinExpr 的 quals 链表中，在本书中对于这类条件通常统称为“约束条件”，但在分解连接树的过程中，这部分还需要详细地分为**过滤（Filter）条件**和**连接（Join）条件**。通常而言，出现在 WHERE 子句中的条件是过滤条件，出

非卖品，仅供非商业用途或交流学习使用

现在 ON 子句中的条件是连接条件，但是随着约束条件的下推，能下推的连接条件也会转化成过滤条件。

在逻辑重写优化的预处理表达式（preprocess_expression 函数）的过程中对谓词进行了正则处理，PostgreSQL 数据库默认约束条件顶层的形式符合合取范式，也就是 AND 的形式。在逻辑分解优化的过程中，会将这些约束条件分发给具体的 RelOptInfo，同时会将原来的约束条件（表达式）转换成 RestrictInfo 结构体，每个 RestrictInfo 对应一个约束条件。

```
typedef struct RestrictInfo
{
    NodeTag    type;
    Expr       *clause;    //一项约束条件

    //该条件是下推条件还是在原始的语法位置，用来标识过滤条件和连接条件
    bool       is_pushed_down;
    bool       outerjoin_delayed;    //该条件在下推过程中是否被阻碍
    bool       can_join;    //如果是 Var OP Var 的形式，则可能被用于 Merge Join 或者 Hash Join
    bool       pseudoconstant; //这是一个常量表达式

    bool       leakproof;        //安全相关参数
    Index       security_level; //安全相关参数
    Relids       clause_relids; //约束条件中引用的表的 rtindex 集合
    Relids       required_relids; //约束条件被应用的最小集合
    Relids       outer_relids; //约束条件所在层次的 Nonnullable-side 的表的 rtindex 集合
    Relids       nullable_relids;//约束条件下层外连接的 Nullable-side 的表的集合

    //如果是操作符表达式，而且有两个参数，
    //分别记录操作符两端的表达式引用的表的 left_relids 和 right_relids
    Relids       left_relids;//左操作数对应的表
    Relids       right_relids;//右操作数对应的表

    //对于 Clause1 OR Clause2 这种约束条件的情况会生成两个 RestrictInfo
    //每个 RestrictInfo 的 clause 分别记录的是 Clause1 和 Clause2
    //然后将两个 RestrictInfo 合成一个 orclause，存放到一个新的 RestrictInfo 中
    Expr       *orclause;
     EquivalenceClass *parent_ec;    /* generating EquivalenceClass */

    /* cache space for cost and selectivity */
    QualCost    eval_cost;        //应用该条件需要的代价
    Selectivity norm_selec;    //内连接对应的选择率
    Selectivity outer_selec;    //外连接对应的选择率
```


非卖品，仅供非商业用途或交流学习使用

```
    //操作符族，适用于Merge Join
    List      *mergeopfamilies;

    //如果适用于Merge Join，则初始化下面的值
    EquivalenceClass *left_ec;    //左操作数对应的等价类
    EquivalenceClass *right_ec;   //右操作数对应的等价类
    EquivalenceMember *left_em; //左操作数生成等价类成员
    EquivalenceMember *right_em; //右操作数生成等价类成员

    //应用这个约束条件做Merge Join的时候，每次计算的选择率都保存下来
    List      *scansel_cache;

    //当使用约束条件对表做连接操作时，是否left_relids中的表作为外表
    bool       outer_is_left;    /* T = outer var on left, F = on right */

    //约束条件适用于Hash Join，将Hash Join的操作符Oid记录在这里
    Oid          hashjoinoperator;    /* copy of clause operator */

    //Hash Join情况下估计的桶数
    Selectivity left_bucketsize;    //如果left_relids是内表
    Selectivity right_bucketsize;    //如果right_relids是内表
} RestrictInfo;
```

在当前阶段，make_restrictinfo 不会把 RestrictInfo 结构体中的所有变量都填充完，我们下面分析 make_restrictinfo 函数的实现流程。

```
RestrictInfo *
make_restrictinfo(……)
{
    //PostgreSQL的约束条件默认是合取范式，这里不会出现AND类型的约束条件
    //如果是or子句，那么就调用make_sub_restrictinfos函数递归处理
    if (or_clause((Node *) clause))
        return (RestrictInfo *) make_sub_restrictinfos(……);

    //不可能出现AND子句
    Assert(!and_clause((Node *) clause));

    //如果是一个单独的表达式，那么直接生成RestrictInfo
    return make_restrictinfo_internal(……);
}
```


非卖品，仅供非商业用途或交流学习使用

在 make_sub_restrictinfos 函数中，需要递归处理 3 种情况：析取范式、合取范式和单独的表达式。

如果是析取范式，则针对析取范式中的每个子约束条件生成一个子 RestrictInfo，并将这些子 RestrictInfo 保存在链表中，然后生成一个父 RestrictInfo，将子 RestrictInfo 链表保存在父 RestrictInfo 中的 orclause 变量中。例如对于 (A OR B OR C)这样的析取范式，首先会分别给 A、B、C 生成 3 个 RestrictInfo，并将这 3 个 RestrictInfo 保存在一个链表中，然后调用 make_restrictinfo_internal 函数，并以这个链表为参数创建一个总的 RestrictInfo，如图 4-10 所示。

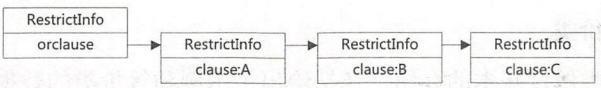


图 4-10　约束条件内存示意图

如果是合取范式，则针对合取范式中的每个子句生成一个子 RestrictInfo，并将这些子 RestrictInfo 保存在链表中，然后调用 make_clause 函数生成一个 BoolExpr 表达式，BoolExpr 表达式的参数就是子 RestrictInfo 的链表。例如对于(A AND B AND C)这样的合取范式，首先分别给 A、B、C 生成 3 个 RestrictInfo，并将这 3 个 RestrictInfo 保存在一个链表中，然后调用 make_clause 函数生成 BoolExpr 表达式，如图 4-11 所示。

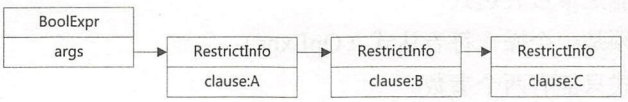


图 4-11　约束条件内存示意图

如果是单独的表达式，则直接调用 make_restrictinfo_internal 函数生成一个 RestrictInfo。

我们再构建一个略为复杂的用例，例如(A OR B OR (C AND D))对应的内存结构如图 4-12 所示。

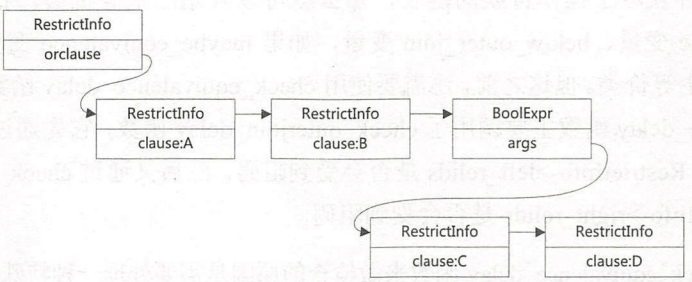


图 4-12　约束条件内存示意图

非卖品，仅供非商业用途或交流学习使用

4.3.6.7 添加 Var 到 targetlist

在约束条件中会涉及一些表上的列，在查询计划的执行阶段，这些列需要被求值，然后判断约束条件的真或假，为了方便使用这些值，需要将这些列增加到 RelOptInfo->reltarget 中，另外，引用这个列属性的约束条件引用到了哪些表，会被存放到 RelOptInfo->attr_needed [Var->attno – RelOptInfo->min_attr]（注意 min_attr 可能是负数，伪列）。

如果发现了 PlaceHolderVar，那么将约束条件引用的表的 rtindex 集合保存在 PlaceHolderVar->ph_needed 中。

4.3.6.8 生成等价类

我们已经对等价类做过基本的介绍，包括给出了示例和等价类的数据结构，现在开始从源代码的角度分析等价类的实现。

一个约束条件是否可以用于生成等价类取决于这个约束条件是否适用于 Merge Join，也就是说这个约束条件必须是 MergeJoinable 的，distribute_qual_to_rels 函数才会考虑将这个约束条件进行生成等价类的处理。

那么什么样的约束条件是 MergeJoinable 的呢？它需要满足以下条件。

- 约束条件不能是常量表达式。
- 约束条件必须是一个操作符表达式（OpExpr）。
- 操作符表达式只能有两个参数。
- 操作符必须是 MergeJoinable 的：Form_pg_operator-> oprcanmerge == true。
- 不能包含易失性函数（Volatile Function）。

MergeJoinable 的条件判断在 check_mergejoinable 函数中实现，如果满足 MergeJoinable 条件，这个函数还会获得对应的操作符族。

如果约束条件获取了操作符族的链表，那么就可以开始产生等价类，我们已经准备好了 maybe_equivalence 变量、below_outer_join 变量，如果 maybe_equivalence 变量设置为 true，就代表可以考虑产生等价类。但这之前，还需要使用 check_equivalence_delay 函数再做一次检查，check_equivalence_delay 函数主要调用了 check_outerjoin_delay 函数，它先通过 check_outerjoin_delay 函数检查了 RestrictInfo->left_relids 是否会受到阻碍，然后又通过 check_outerjoin_delay 函数检查了 RestrictInfo->right_relids 是否会受到阻碍。

这里增加 check_equivalence_delay 函数来做检查的原因是需要处理一种特殊情况，例如对于语句 SELECT * FROM STUDENT LEFT JOIN SCORE ON TRUE WHERE COALESCE(SCORE.sno, 1)

非卖品，仅供非商业用途或交流学习使用

= STUDENT.sno AND STUDENT.sno = COALESCE(SCORE.cno, 20)，其中 COALESCE(SCORE.sno, 1) = STUDENT.sno AND STUDENT.sno = COALESCE(SCORE.cno, 20)，这两个约束条件由于使用了 COALESCE 函数，导致它是不严格的，因此无法消除外连接，它原本是不能下推的。

但是仔细分析发现，在使用 check_outerjoin_delay 函数进行检查的时候，这个约束条件没有受到阻碍，也就是说 maybe_equivalence == true，这时候如果不调用 check_equivalence_delay 函数的检查，就会直接进入 process_equivalence 函数生成等价类。

```
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN SCORE ON TRUE WHERE COALESCE(SCORE.sno,
1) = STUDENT.sno AND STUDENT.sno = COALESCE(SCORE.cno, 20);
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..2.21 rows=1 width=23)
   Filter: ((COALESCE(score.sno, 1) = student.sno) AND (student.sno = COALESCE(score.cno,
20)))
   ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
   ->  Materialize  (cost=0.00..1.01 rows=1 width=12)
         ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
(5 rows)
```

如果在源代码中去掉 check_equivalence_delay 函数，查看同一 SQL 语句的执行计划发现，基于 WHERE COALESCE(SCORE.sno, 1) = STUDENT.sno AND STUDENT.sno = COALESCE (SCORE.cno, 20)这个约束条件产生的等价类，推理出了新的约束条件(COALESCE(sno, 1) = COALESCE(cno, 20))，这个推理出来的新条件下推到了 SCORE 表上。

```
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN SCORE ON TRUE WHERE COALESCE(SCORE.sno,
1) = STUDENT.sno AND STUDENT.sno = COALESCE(SCORE.cno, 20);
                        QUERY PLAN
-----------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..2.19 rows=1 width=23)
   Filter: (student.sno = COALESCE(score.sno, 1))
   ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
   ->  Materialize  (cost=0.00..1.02 rows=1 width=12)
         ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
               Filter: (COALESCE(sno, 1) = COALESCE(cno, 20))
(6 rows)
```

如果 maybe_equivalence 变量的值是 true，且通过了 check_equivalence_delay 的检查，那么就可以基于当前的约束条件生成等价类（process_equivalence 函数）。

首先需要查找当前已经存在的等价类，看一下当前的约束条件两端的表达式是否已经存在

非卖品，仅供非商业用途或交流学习使用

于等价类中。

```
//在已经存在的等价类中查找，看当前约束条件的两个表达式是否已经存在于某个等价类之中
ec1 = ec2 = NULL;
em1 = em2 = NULL;
foreach(lc1, root->eq_classes)
{
   EquivalenceClass *cur_ec = (EquivalenceClass *) lfirst(lc1);
   ListCell  *lc2;

//跳过含有易失性函数的等价类
//如果排序校正方式不同，则跳过
//如果不属于同一个操作符族，也跳过

   //查找匹配等价类中的每一个成员
   foreach(lc2, cur_ec->ec_members)
   {
      EquivalenceMember *cur_em = (EquivalenceMember *) lfirst(lc2);
      ......

      //item1 是约束条件的一个表达式，如果找到了，记录等价类和等价类成员
      if (!ec1 &&
         item1_type == cur_em->em_datatype &&
         equal(item1, cur_em->em_expr))
      {
         ec1 = cur_ec;
         em1 = cur_em;
         if (ec2)
            break;
      }

      //item2 是约束条件的另一个表达式，如果找到了，记录等价类和等价类成员
      if (!ec2 &&
         item2_type == cur_em->em_datatype &&
         equal(item2, cur_em->em_expr))
      {
         ec2 = cur_ec;
         em2 = cur_em;
         if (ec1)
            break;
      }
```

```
    }

//只有约束条件两端的表达式全部匹配上或者全部的等价类被遍历结束，才会结束查找
    if (ec1 && ec2)
        break;
}
```

在查找之后，分成了 4 种情况。

- 情况 1：约束条件两端的表达式在同一个等价类中已经存在，这种情况需要在 RestrictInfo 中补充等价类信息。

```
if (ec1 == ec2)
{
    //将当前约束条件（RestrictInfo）加入 source
    ec1->ec_sources = lappend(ec1->ec_sources, restrictinfo);

    //当前约束条件是否处在外连接之下
    ec1->ec_below_outer_join |= below_outer_join;
    ……

    //将等价类信息记录到 RestrictInfo 中
    restrictinfo->left_ec = ec1; //因为在同一个等价类中，left_ec 和 right_ec 相同
    restrictinfo->right_ec = ec1;
    /* mark the RI as usable with this pair of EMs */
    restrictinfo->left_em = em1; //记录对应的等价类成员
    restrictinfo->right_em = em2;//记录对应的等价类成员
    return true;
}
```

- 情况 2：约束条件两端的表达式均被找到，但是不属于同一个等价类，这种情况可以直接合并等价类。

```
//合并两个等价类，包括 ec_members、ec_sources、ec_derives 等
ec1->ec_members = list_concat(ec1->ec_members, ec2->ec_members);
ec1->ec_sources = list_concat(ec1->ec_sources, ec2->ec_sources);
ec1->ec_derives = list_concat(ec1->ec_derives, ec2->ec_derives);
ec1->ec_relids = bms_join(ec1->ec_relids, ec2->ec_relids);
ec1->ec_has_const |= ec2->ec_has_const;
/* can't need to set has_volatile */
ec1->ec_below_outer_join |= ec2->ec_below_outer_join;
……
```

```
//记录合并链，表示 ec2 已经被 ec1 合并
ec2->ec_merged = ec1;
//在等价类列表中删除 ec2
root->eq_classes = list_delete_ptr(root->eq_classes, ec2);
......

//在 ec1 中增加和当前约束条件相关的信息
ec1->ec_sources = lappend(ec1->ec_sources, restrictinfo);
ec1->ec_below_outer_join |= below_outer_join;
......

//ec1 和 ec2 这两个等价类的成员都合并到 ec1 中
restrictinfo->left_ec = ec1;
restrictinfo->right_ec = ec1;
restrictinfo->left_em = em1;
restrictinfo->right_em = em2;
```

- 情况 3：约束条件左端的表达式在某个等价类中找到，右端的表达式没有找到，这种情况需要将右端的表达式加入左端表达式所在的等价类中，或者，约束条件右端的表达式在某个等价类中找到，左端的表达式没有找到，这种情况需要将左端的表达式加入右端表达式所在的等价类中，这部分的源代码实现和上面两种情况的实现大同小异。
- 情况 4：约束条件两端的表达式在现有的等价类中都没有找到，这时候需要新建一个拥有两个成员的等价类，这部分的源代码实现和上面两种情况的实现大同小异。

需要注意的是，在生成等价类之后，当前的约束条件并没有真正分配给 RelOptInfo，从源代码中可以看出这里生成等价类之后直接返回了，并没有真正进行分发。

如果 check_equivalence_delay 函数失败，则代表当前的约束条件两端的表达式是不等价的，但仍然不妨碍给每个表达式生成一个单成员的等价类。这种等价类有两个作用：一个作用是在后续的等价推理过程中，仍然可以尝试对这种等价类进行合并；另一个作用是对指定了排序关键字的列，会生成一个 PathKeys 来记录排序的顺序，单成员的等价类可以用来构建这个 PathKeys，因此单成员的等价类也是有用的。

生成单成员的等价类是在 initialize_mergeclause_eclasses 函数中实现的，initialize_mergeclause_eclasses 又调用了 get_eclass_for_sort_expr 函数来真正地生成等价类，get_eclass_for_sort_expr 函数首先查找当前的等价类列表（PlannerInfo->eq_classes），如果能匹配到当前的表达式，那么就直接返回这个等价类，如果在已经存在的等价类中匹配不到，则生成一个新的等价类，并存放到 PlannerInfo->eq_classes 中，单成员的等价类生成后还会分别记录在

RestrictInfo->left_ec 和 RestrictInfo->right_ec 中。

```
//可以生成等价类，去尝试生成等价类
if (maybe_equivalence)
{
   //在生成等价类之前，再做一次检查，过滤掉一些特殊情况
   if (check_equivalence_delay(root, restrictinfo) &&
      //生成等价类，先查找已经存在的等价类，
      //如果能找到就合并等价类，不能找到则创建新的等价类
      process_equivalence(root, restrictinfo, below_outer_join))
      //***注意，这里直接 return，约束条件暂时不会分发到 RelOptInfo 上
      return;

   //如果 check_equivalence_delay 函数的检查没有通过，则创建单成员的等价类
   initialize_mergeclause_eclasses(root, restrictinfo);
   //***注意：如果执行到这里，那么就会去分发约束条件到 RelOptInfo
}
```

对于 maybe_equivalence 变量为 false 的情况，就有可能是外连接（maybe_outer_join == true，以及 RestrictInfo->can_join），PostgreSQL 数据库对外连接的情况还需要继续进行处理，因此也对这种约束条件生成单成员的等价类。

```
//对于这种情况，到 reconsider_outer_join_clauses 函数中还会去处理
else if (maybe_outer_join && restrictinfo->can_join) //外连接导致的不可推理
{
   //生成单成员等价类
   initialize_mergeclause_eclasses(root, restrictinfo);

   //如果 Nonnullable-side 的表的列在约束条件的左边，
   //将约束条件记录到 PlannerInfo->left_join_clauses
   if (bms_is_subset(restrictinfo->left_relids,
                     outerjoin_nonnullable) &&
      !bms_overlap(restrictinfo->right_relids,
                   outerjoin_nonnullable))
   {
      /* we have outervar = innervar */
      root->left_join_clauses = lappend(root->left_join_clauses,
                                        restrictinfo);
      return;
   }

   //如果 Nonnullable-side 的表的列在约束条件的右边，
```

```
    //将约束条件记录到 PlannerInfo->right_join_clauses
    if (bms_is_subset(restrictinfo->right_relids,
                      outerjoin_nonnullable) &&
        !bms_overlap(restrictinfo->left_relids,
                     outerjoin_nonnullable))
    {
        /* we have innervar = outervar */
        root->right_join_clauses = lappend(root->right_join_clauses,
                                           restrictinfo);
        return;
    }

    //如果是 Full Join，记录到 PlannerInfo->full_join_clauses
    if (jointype == JOIN_FULL)
    {
        /* FULL JOIN (above tests cannot match in this case) */
        root->full_join_clauses = lappend(root->full_join_clauses,
                                          restrictinfo);
        return;
}

//***注意，如果约束条件记录到了 left_join_clauses 或 right_join_clauses
//或 full_join_clauses，这种情况不会进入分发约束条件给 RelOptInfo
}
```

另外，对于其他情况则直接生成单成员的等价类。

```
else
{
    //生成单成员的等价类
initialize_mergeclause_eclasses(root, restrictinfo);
//***注意，这种情况会进入分发约束条件给 RelOptInfo
}
```

4.3.6.9 分发约束条件给 RelOptInfo

有些约束条件（RestrictInfo）被记录到了等价类中，有些则被记录到了 PlannerInfo 中，这些约束条件在目前的阶段不分发给 RelOptInfo，但是对于其他生成单成员等价类的约束条件，需要对其进行分发，分发是在 distribute_restrictinfo_to_rels 函数中实现的。

目前 RestrictInfo->required_relids 中记录了约束条件要被应用的时候会应用到哪些表，这时候分成两种情况。

- 如果 RestrictInfo->required_relids 中只有一个表，那可以把这个约束条件存放到 RelOptInfo->baserestrictinfo 中。
- 如果 RestrictInfo->required_relids 中多于一个表，需要检查一下这个约束条件的操作符是否能够满足 Hash Join 的要求（Form_pg_operator-> oprcanhash），另外，把这个约束条件存放到对应的表的 RelOptInfo-> joininfo 中。

存放在 RelOptInfo->joininfo 中的约束条件可能是连接条件，也可能是过滤条件，在生成连接路径的时候会根据 RestrictInfo-> is_pushed_down 来进行区分。

4.3.7　reconsider_outer_join_clauses 函数

在生成等价类的阶段，有一些条件没有真正下推下去，而是记录到了 PlannerInfo 中的 left_join_clauses、right_join_clauses、full_join_clauses 中，这些条件当时没有下发下去是因为还有优化的空间，所以还需要 reconsider_outer_join_clauses 函数来做“reconsider”。

reconsider_outer_join_clauses 函数又考虑了两种情况，一种情况是左外连接的优化，另一种情况是全连接的优化。

情况 1：如果查询语句是左外连接，而且有这样一个等价类{Var of Nonnullable -side, CONST}，也就是 {外表的列，常量} 这样的等价类，另外连接条件中包含 Var of Nonullable-side = Var of Nullable-side 的情况，那么就可以产生一个 Var of Nullable-side = CONST 的过滤条件，下推到 Nullable-side 的表上，这种情况的约束条件记录在 PlannerInfo->left_join_clauses 中，这里给出一个示例。

```
--情况 1.1，左外连接的优化，RestrictInfo 记录在 PlannerInfo->left_join_clauses 中
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN SCORE ON STUDENT.sno = SCORE.sno WHERE
STUDENT.sno=1;
                        QUERY PLAN
------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..2.11 rows=1 width=23)
   Join Filter: (student.sno = score.sno)
   -> Seq Scan on student  (cost=0.00..1.09 rows=1 width=11)
        Filter: (sno = 1)
   -> Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
        Filter: (sno = 1)
(6 rows)
```

如果交换一下 STUDENT.sno 和 SCORE.sno 的顺序，就构成了情况 1 的另一种形式。

```
--情况 1.2，左外连接的优化，RestrictInfo 记录在 PlannerInfo->left_join_clauses 中
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN SCORE ON SCORE.sno = STUDENT.sno WHERE
STUDENT.sno=1;
                        QUERY PLAN
------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..2.11 rows=1 width=23)
   Join Filter: (score.sno = student.sno)
   -> Seq Scan on student  (cost=0.00..1.09 rows=1 width=11)
        Filter: (sno = 1)
   -> Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
        Filter: (sno = 1)
(6 rows)
```

情况 2：全外连接的优化，如果有一个连接条件 Var of LHS = Var of RHS，并且有一个过滤条件 COALESCE(Var of LHS, Var of RHS) = CONST，这样就能构成两个过滤条件 Var of LHS = CONST 和 Var of RHS = CONST 分别下推到对应的表上。

```
--情况 2，全外连接的优化
postgres=# EXPLAIN SELECT * FROM STUDENT FULL JOIN SCORE ON STUDENT.sno = SCORE.sno WHERE
COALESCE(STUDENT.sno, SCORE.sno) = 1;
                        QUERY PLAN
------------------------------------------------------------------
 Hash Full Join  (cost=1.02..2.13 rows=1 width=23)
   Hash Cond: (student.sno = score.sno)
   -> Seq Scan on student  (cost=0.00..1.09 rows=1 width=11)
        Filter: (sno = 1)
   -> Hash  (cost=1.01..1.01 rows=1 width=12)
        -> Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
             Filter: (sno = 1)
(7 rows)
```

reconsider_outer_join_clauses 函数目前主要处理的就是以上两种情况，下面根据这两种情况分析一下源代码。

```
void
reconsider_outer_join_clauses(PlannerInfo *root)
{
   do
   {
      //情况 1.1
```

```
        for (cell = list_head(root->left_join_clauses); cell; cell = next)
        { ····· }

        //情况 1.2
        for (cell = list_head(root->right_join_clauses); cell; cell = next)
        { ····· }

        //情况 2:
        for (cell = list_head(root->full_join_clauses); cell; cell = next)
        { ····· }
    } while (found);

    //distribute_qual_to_rels 函数有些约束条件（RestrictInfo）没有分发
    //在这里负责分发
    foreach(cell, root->left_join_clauses)
    {
        RestrictInfo *rinfo = (RestrictInfo *) lfirst(cell);
        distribute_restrictinfo_to_rels(root, rinfo);
    }
    ·····
}
```

4.3.7.1　左外连接的优化

情况 1.1 和情况 1.2 属于左外连接的优化，这两种情况优化的方法是相同的，都是调用了 reconsider_outer_join_clause 函数实现的，只不过在参数上略有区分，outer_on_left 参数如果是 true，则代表是情况 1.1，如果是 false，则代表的是情况 1.2。

对于左外连接优化首先要做的就是在 PlannerInfo->left_join_clauses 中查找符合 Var of Nonullable-side = Var of Nullable-side 的连接条件。

```
//情况 1.1
if (outer_on_left)
{
    //外表（Nonnullable-side）的列标量在左边，内表的列表量在右边
    outervar = (Expr *) get_leftop(rinfo->clause);
    innervar = (Expr *) get_rightop(rinfo->clause);
    inner_datatype = right_type;
    inner_relids = rinfo->right_relids;
}
//情况 1.2
else
```

```
{
   //外表（Nonnullable-side）的列标量在右边，内表的列表量在左边
   outervar = (Expr *) get_rightop(rinfo->clause);
   innervar = (Expr *) get_leftop(rinfo->clause);
   inner_datatype = left_type;
   inner_relids = rinfo->left_relids;
}
```

如果有 Var of Nonullable-side = Var of Nullable-side 这样的连接条件，那么就可以查看是否有符合{Var of Nonnullable -side, CONST}的等价类。在 reconsider_outer_join_clause 函数中，等价类的检查主要检查了两项，一个是有没有一个等价类是包含 Var 和 CONST 常量的，另一个是等价类中的 Var 是不是 Nonnullable-side 的表的 Var。

```
foreach(lc1, root->eq_classes)
{
   EquivalenceClass *cur_ec = (EquivalenceClass *) lfirst(lc1);

   //如果等价类中没有常量，就跳过去，因为只关心有常量的等价类
   if (!cur_ec->ec_has_const)
      continue;
   ......
   //找一找外表（Nonnullable-side）的 Var 是不是在这个有常量的等价类里
   match = false;
   //遍历这个有常量的等价类的所有成员
   foreach(lc2, cur_ec->ec_members)
   {
      //如果外表（Nonnullable-side）的 Var 和等价类的一个成员匹配上了
      if (equal(outervar, cur_em->em_expr))
      {
         match = true;//匹配上了
         break;
      }
   }
   //没匹配上，换下一个等价类
   if (!match)
      continue;          /* no match, so ignore this EC */

   ......
}
```

如果两个条件都能满足，那么就开始生成一个新的约束条件（Var of Nullable-side = CONST），

生成新约束条件是在 build_implied_join_equality 函数中通过两个步骤生成的，一个步骤是生成一个表达式，另一个步骤是将表达式转换为另一个形式（RestrictInfo）。

4.3.7.2　全外连接的优化

全外连接的优化主要是调用了 reconsider_full_join_clause 函数实现的，它主要检查是否存在一个连接条件 Var of LHS = Var of RHS 和一个过滤条件 COALESCE(Var of LHS, Var of RHS) = CONST。

```
match = false;
foreach(lc2, cur_ec->ec_members)
{
   coal_em = (EquivalenceMember *) lfirst(lc2);

   //等价类中有一个成员是 COALESCE 表达式
   if (IsA(coal_em->em_expr, CoalesceExpr))
   {
      CoalesceExpr *cexpr = (CoalesceExpr *) coal_em->em_expr;
      Node      *cfirst;
      Node      *csecond;

      //COALESCE 表达式有两个参数
      if (list_length(cexpr->args) != 2)
         continue;

      //获得 COALESCE 表达式的两个参数
      cfirst = (Node *) linitial(cexpr->args);
      csecond = (Node *) lsecond(cexpr->args);

      //这两个参数恰好和约束条件（RestrictInfo）中的两个 Var 一致
      if (equal(leftvar, cfirst) && equal(rightvar, csecond))
      {
         match = true;
         break;
      }
   }
}
```

匹配上了 COALESCE(Var of Nonnullable-side,Var of Nullable-side) = CONST 的情况之后，就可以构建两个新的下推条件：Var of Nonnullable-side = CONST 和 Var of Nullable-side = CONST，构建约束条件仍然是调用了 build_implied_join_equality 函数。

版权所有，严禁传播，违者自负法律责任！

4.3.8 generate_base_implied_equalities 函数

除了在 PlannerInfo 中的 left_join_clauses、right_join_clauses、full_join_clauses 中记录了外连接相关的不能下推的约束条件之外，能够生成等价类的约束条件也没有下推，这些约束条件（RestrictInfo）就记录在等价类的 EquivalenceClass->ec_sources 中，generate_base_implied_equalities 函数尝试对其再做一些优化。

generate_base_implied_equalities 函数的流程如图 4-13 所示，它主要处理两种情况，一种是处理含有 CONST 的等价类，另一种是同一个表的两个列出现在等价类中的情况。

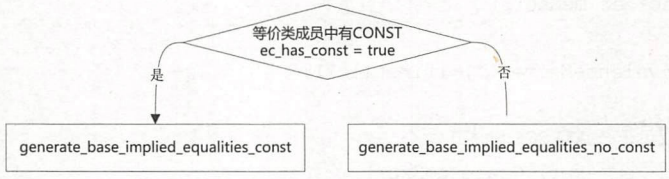


图 4-13 等价类推理流程

4.3.8.1 带有常量的情况

如果一个等价类中有常量，那么：

- 找到等价类中的第一个常量。
- 用等价类中的其他成员和这个常量构成约束条件（如果其他成员也是常量，则通过 eval_const_expressions 对这个 CONST OP CONST 的约束条件化简）。
- 将生成的约束条件下推到对应的表（RelOptInfo）上。

例如对于 SQL 语句 SELECT * FROM STUDENT,SCORE WHERE STUDENT.sno = SCORE.sno AND STUDENT.sno = 1，它会建立{STUDENT.sno, SCORE.sno , 1}这样一个有 3 个成员的等价类，generate_base_implied_equalities_const 函数就会将等价类中的常量{1}找到，然后生成 STUDENT.sno = 1 和 SCORE.sno = 1 两个约束条件，分别下推到 STUDENT 和 SCORE 两个表上，而约束条件 STUDENT.sno = SCORE.sno 虽然在 SQL 语句中显式地指定了，通过示例可以看到，执行计划中没有出现 STUDENT.sno = SCORE.sno 这个约束条件。

```
postgres=# EXPLAIN SELECT * FROM STUDENT,SCORE WHERE STUDENT.sno = SCORE.sno AND STUDENT.sno
= 1;
                        QUERY PLAN
-----------------------------------------------------------
 Nested Loop  (cost=0.00..2.11 rows=1 width=23)
   ->  Seq Scan on student  (cost=0.00..1.09 rows=1 width=11)
```

版权所有，严禁传播，违者自负法律责任！


```
      Filter: (sno = 1)
  ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
      Filter: (sno = 1)
(5 rows)
```

4.3.8.2　单表列属性的情况

如果一个等价类中出现了同一个表上的两个列，那么就可以构成一个单表的等值条件，然后下推到对应的表上，对于 SQL 语句 SELECT *FROM STUDENT,SCORE WHERE STUDENT.sno = SCORE.sno AND SCORE.cno = STUDENT.sno，约束条件可以构成一个等价类{STUDENT.sno, SCORE.sno，SCORE.cno}，其中 SCORE.sno 和 SCORE.cno 是一个表上的列，因此可以构成一个约束条件下推到表上。

```
postgres=# EXPLAIN SELECT *FROM STUDENT,SCORE WHERE STUDENT.sno = SCORE.sno AND SCORE.cno
= STUDENT.sno;
                        QUERY PLAN
-----------------------------------------------------------------------
 Hash Join  (cost=1.02..2.13 rows=1 width=23)
   Hash Cond: (student.sno = score.sno)
   ->  Seq Scan on student  (cost=0.00..1.07 rows=7 width=11)
   ->  Hash  (cost=1.01..1.01 rows=1 width=12)
       ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=12)
           Filter: (sno = cno) <- SCORE.sno = SCORE.cno
(6 rows)
```

4.3.9　记录表之间的等价关系

至此，所有“能下推”的 RestrictInfo 都已经“下推”完毕，其中大部分是 SQL 语句中显式指定的约束条件，也有部分（reconsider_outer_join_clauses 函数和 generate_base_implied_equalities 函数中生成的部分）是根据等价类和等价约束条件生成的。

但是还有一些等价类 ec_sources 中的约束条件没有下推，因为根据等价类还能推理出一些其他的连接条件，这些连接条件在生成连接路径的时候才会根据等价类临时生成。目前要做的是，一个等价类中的成员涉及了哪些表记录下来，这样在生成执行路径的时候，就可以根据这个表上有没有等价类来确定是不是可能根据等价类生成连接条件，从而产生连接关系，表之间的等价类关系通过 has_relevant_eclass_joinclause 函数来判断。

版权所有，严禁传播，违者自负法律责任！

4.4 PlaceHolderVar 的作用

我们虽然多次提及了 PlaceHolderVar，在 PostgreSQL 数据库查询优化器的源代码中也多次用到了 PlaceHolderVar，但一直没有做详细的说明，现在应该是时候对 PlaceHolderVar 做一个总结性的说明了。

从查询优化器的代码可以看出，PlaceHolderVar 总是和 Var 成对出现，看上去和 Var 并没有太多的分别，实际上 PlaceHolderVar 就是一种特殊的 Var，它是对 Var 功能的一种“增强”。我们在查询优化的过程中需要对查询树进行基于逻辑的等价变换，变换的过程中需要调整子查询的位置、调整 Var 变量的位置或者消减掉一些不必要的操作。在有些情况下逻辑优化会受到限制，例如调整 Var 变量的位置之后查询树逻辑上就不等价了，这时候就需要找到一个办法，即能调整 Var 变量的位置，还要保证逻辑等价，PlaceHolderVar 就应运而生了。

我们在子查询提升的部分已经简要地介绍了 PlaceHolderVar 的使用，不妨把例子在这里再展示一下。

```
postgres=# EXPLAIN VERBOSE SELECT * FROM STUDENT st LEFT JOIN (SELECT sno, COALESCE(degree,60)
FROM SCORE) sc ON st.sno=sc.sno;
                          QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..250.77 rows=71 width=19)
   Output: st.sno, st.sname, st.ssex, score.sno, (COALESCE(score.degree, 60))
   Join Filter: (st.sno = score.sno)
   -> Seq Scan on public.student st  (cost=0.00..1.07 rows=7 width=11)
        Output: st.sno, st.sname, st.ssex
   -> Materialize  (cost=0.00..40.60 rows=2040 width=8)
        Output: score.sno, (COALESCE(score.degree, 60))
        -> Seq Scan on public.score  (cost=0.00..30.40 rows=2040 width=8)
             Output: score.sno, COALESCE(score.degree, 60)
(9 rows)
```

在示例中的子查询提升的过程中，会将子查询中的 COALESCE(degree,60)也提升一层，但是由于：

- SQL 语句本身是左外连接语句。
- COALESCE(degree,60)是不严格的。
- degree 处于左外连接的 Nullable-side。

如果直接调整表达式 COALESCE(degree,60)的位置，查询树在逻辑上就不等价了，因此表

版权所有，严禁传播，违者自负法律责任！

达式 COALESCE(degree,60)还必须在下层执行，子查询也就无法提升了。

如果既想获得子查询提升带来好处，又想对查询树做逻辑上的等价变换，可以尝试这样做：

- 子查询仍然提升。
- 将表达式 COALESCE(degree,60)变成 PlaceHolderVar。

一旦把表达式 COALESCE(degree,60)转换成 PlaceHolderVar，就等于给 COALESCE(degree,60)做上了标记，它就能告知后续的查询优化操作有这样一种类型的“Var”，它需要做这样的特殊处理，它在对表 SCORE 做扫描的时候直接对投影列 degree 做 COALESCE(degree,60)运算，在输出 STUDENT 表和 SCORE 表的连接结果（上层）的时候直接显示运算结果，实际的效果就类似子查询已经提升，但是表达式 COALESCE(degree,60)的求值还是在表扫描的阶段（下层）进行求值。

现在我们总结一下有哪些情况可能需要使用 PlaceHolderVar。

- 查询语句中必须有层次关系，也就是说在有子查询或者子连接、继承表、UNION 操作产生 AppendRel 的情况下（我们把这些称为下层查询）才可能需要将 Var 或者表达式封装成 PlaceHolderVar。
- 查询语句中如果有外连接，且下层查询处于 Nullable-side，且表达式是不严格的（注意，我们在第 3 章外连接消除时将严格地定义分成了精确的严格和宽泛的严格，这里的严格是精确的严格，也就是说输入参数是 NULL 值，输出结果是 NULL 值，如果输入参数是NULL值，输出结果是FALSE，不在这次处理的范围内），那么会封装成PlaceHolderVar。
- 如果 AppendRel 中出现了表达式（也就是说不是简单的 Var），那么一定会封装成 PlaceHolderVar。
- 如果下层查询引用了上层查询表中的属性（例如 Lateral），那么会将这个属性或属性相关的表达式封装成 PlaceHolderVar。

我们来看一下 PlaceHolderVar 的结构。

```
typedef struct PlaceHolderVar
{
    Expr    xpr;
    Expr   *phexpr;             //被封装的Var或者表达式
    Relids   phrels;            //子查询的表对应的rtindex
    Index       phid;           //PlannerGlobal->lastPHId负责生成唯一的ID给每个PlaceHolderVar
    Index       phlevelsup;     //和Var的varlevelsup的功能相同
} PlaceHolderVar;
```

版权所有，严禁传播，违者自负法律责任！

替换的主要操作在 pullup_replace_vars 函数中实现，它通过几个变量来控制是否需要进行替换，需要关注的是 pullup_replace_vars 函数的 pullup_replace_vars_context 类型的参数，pullup_replace_vars_context 结构体中有两个和 PlaceHolderVar 相关的变量。

- pullup_replace_vars_context->need_phvs：它的作用是“粗略”地判断是否需要使用 PlaceHolderVar 进行封装，如果在外连接之下或者包含 Appendrel 子表的情况下，“可能”是需要使用 PlaceHolderVar 进行封装的。
- pullup_replace_vars_context-> wrap_non_vars：它的作用是指明下层查询中只要出现了表达式，就使用 PlaceHolderVar 进行封装，目前主要针对 Appendrel 子表下的表达式。

另外在递归处理函数 pullup_replace_vars_callback 中对各种情况进行了细化（可以结合上述说明来查看源代码分析）。

```
//need_phvs 是粗略的判断，还需要具体的细化
if (rcon->need_phvs)
{
   bool       wrap; //wrap 代表是否使用 PlaceHolderVar 进行封装

   if (newnode && IsA(newnode, Var) &&
      ((Var *) newnode)->varlevelsup == 0)
   {
      //如果是一个普通的 Var，
      //通常是不需要封装的(即使 need_phvs == true，因为 need_phvs 是粗略判断)
      //但如果是 Lateral 变量，那么是需要封装的
      if (rcon->target_rte->lateral &&
         !bms_is_member(((Var *) newnode)->varno, rcon->relids))
      ......
   }
   else if (newnode && IsA(newnode, PlaceHolderVar) &&
         ((PlaceHolderVar *) newnode)->phlevelsup == 0)
   {
      wrap = false; //已经是 PlaceHolderVar 了，不用封装了
   }
   else if (rcon->wrap_non_vars)
   {
      //到这里一定不是简单的 Var 了，wrap_non_vars 代表必须封装非简单的 Var
      wrap = true;
   }
   else
   {
```

版权所有，严禁传播，违者自负法律责任！


```
        //到这里一定不是简单 Var 了
        //如果是 Lateral 子查询，表达式引用了上层的 Var，则需要封装
        if ((rcon->target_rte->lateral ?
            bms_overlap(pull_varnos((Node *) newnode), rcon->relids) :

            //如果不是 Lateral 子查询，表达式引用了当前层的 Var，
            //但表达式不是严格的，需要封装
            contain_vars_of_level((Node *) newnode, 0)) &&
            !contain_nonstrict_functions((Node *) newnode))
        ......
    }
}
```

这时虽然已经将 Var 或者表达式封装成了 PlaceHolderVar 结构体，但是还是无法实现“表达式在下层求值，在上层做投影”的功能，目前还不知道表达式在哪一层求值，在哪一层直接使用下层求出的值，针对这种情况为每个 PlaceHolderVar 再生成一个 PlaceHolderInfo 结构体，我们看一下这个结构体的定义。

```
typedef struct PlaceHolderInfo
{
    NodeTag       type;
    Index         phid;              //和 PlaceHolderVar 中的 phid 对应
    PlaceHolderVar *ph_var;          //和 PlaceHolderVar 中的 phexpr 对应
    Relids        ph_eval_at;        //表达式在哪里一层求值
    Relids        ph_lateral;        //Var 或者表达式是不是 Lateral 变量
    Relids        ph_needed;         //表达式求解获得的值在哪里被使用
    int32         ph_width;          //与 Var 类似，PlaceHolderVar 也设定宽度
} PlaceHolderInfo;
```

Var 代表的列属性或者表达式一旦被封装成 PlaceHolderVar，它的处理方式大体上和 Var 是相同的，因此在 PostgreSQL 数据库的源代码中经常可以看到 Var 的处理和 PlaceHolderVar 的处理成对地出现，不同的是 PlaceHolderVar 的生存周期只在查询优化器内部，在最终生成执行计划的时候，会将 PlaceHolderVar 结构体消除掉。

4.5　Lateral 语法的支持

在 query_planner 函数中分别调用了 find_lateral_references 函数和 create_lateral_join_info 函数两个函数来处理 Lateral 语法中涉及的变量（Var 或者 PlaceHolderVar），在介绍这两个函数之前，先对 Lateral 语法做一个介绍。

版权所有，严禁传播，违者自负法律责任！

4.5.1 Lateral 的语义分析

PostgreSQL 数据库在 Lateral 实现之前是基于这样一个假设：所有的子查询（不包括子连接）都独立存在，不能互相引用属性。例如：

```
postgres=# EXPLAIN SELECT * FROM STUDENT, (SELECT STUDENT.sno, degree FROM SCORE) sc;
ERROR:  invalid reference to FROM-clause entry for table "student"
```

示例中子查询的投影列中引用了 STUDENT 表的属性，这导致了整个语句执行失败，如果打算执行这样的语句，就需要在子查询前面显式地指定 Lateral 关键字，这样 SQL 语句就能正常执行了。

```
postgres=# EXPLAIN SELECT * FROM STUDENT, Lateral (SELECT STUDENT.sno, degree FROM SCORE)
sc;
                          QUERY PLAN
---------------------------------------------------------------------------
 Nested Loop  (cost=0.00..74167.39 rows=1289119 width=27)
   ->  Seq Scan on score  (cost=0.00..1.01 rows=1 width=4)
   ->  Seq Scan on student  (cost=0.00..61275.19 rows=1289119 width=19)
(3 rows)
```

要支持 Lateral 语义，首先要在语法分析阶段增加语法支持，以 Lateral 的子查询为例，在语法分析文件（gram.y）中的处理如下。

```
| LATERAL_P select_with_parens opt_alias_clause
    {
        RangeSubselect *n = makeNode(RangeSubselect);
        n->lateral = true;
        n->subquery = $2;
        n->alias = $3;
        ......
        $$ = (Node *) n;
    }
```

这样就在 RangeSubselect 结构体中记录了当前的子查询具有 Lateral 性质，在对 RangeSubselect 进行语义分析时，就可以设置 Lateral 标记到 ParseState->p_lateral_active 中，因为 ParseState 是语义分析阶段的上下文句柄，因此可以在语义分析阶段观察到 p_lateral_active 变量的值。

```
pstate->p_lateral_active = r->lateral;//根据 RangeSubselect 设置 lateral 标记

//处理子查询，在子查询处理过程中 ParseState->p_lateral_active == true 就可以“放过”
```

版权所有，严禁传播，违者自负法律责任！


```
//原本报错的表属性
query = parse_sub_analyze(r->subquery, pstate, NULL,
                          isLockedRefname(pstate, r->alias->aliasname),
                          true);
pstate->p_lateral_active = false;//恢复 lateral 标记
```

设置了 ParseState->p_lateral_active 标记之后，在对子查询进行语义分析的过程中，就可以对 STUDENT.sno 这样的列属性进行处理。列属性的语义分析在 transformColumnRef 函数中实现，这个函数的主要功能是将 ColumnRef 结构体转换成 Var 结构体，也就是说把语法树（语法分析阶段生成语法树）中的列属性 ColumnRef 转换成查询树（语义分析阶段生成查询树）中的列属性 Var。

在 ColumnRef 到 Var 的转换过程中，如果没有 Lateral 语法的支持，它只会查 ColumnRef 是否属于本层次的某个表（子查询子句就只看自己子查询中的表），在有了 Lateral 标识之后，放开了限制，还可以查看 ColumnRef 是否还属于上层的表，这是因为 ParseState 是一个链表结构，每个 ParseState 都有一个指向自己父 ParseState 的指针，顶层的 ParseState 的父 ParseState 指针为 NULL。

```
RangeTblEntry * refnameRangeTblEntry(……)
{
   ……
    while (pstate != NULL)
    {
        RangeTblEntry *result;
        // scanNameSpaceForRelid 函数和 scanNameSpaceForRefname 函数中
        // 中都判断了 ParseState->p_lateral_active 标识来决定是否“放行”
        if (OidIsValid(relId))
            result = scanNameSpaceForRelid(pstate, relId, location);
        else
            result = scanNameSpaceForRefname(pstate, refname, location);
        if (result)
            return result;
        ……
        //查看父 ParseState
        pstate = pstate->parentParseState;
    }
    return NULL;
}
```

4.5.2 收集 Lateral 变量

在语义分析阶段主要是对需要 Lateral 的列属性“放行”，对 Lateral 的列属性的处理主要集中在数据库的查询优化阶段。首先要处理的就是在查询树中收集 Lateral 变量，这个工作在 find_lateral_references 函数中实现，find_lateral_references 函数遍历 PlannerInfo 中的 RelOptInfo 数组（PlannerInfo->simple_rel_array），从这些 RelOptInfo 中查找是否有 Lateral 列属性，目前 Lateral 搜索了如下类型的基表。

```
if (rte->rtekind == RTE_RELATION)
    vars = pull_vars_of_level((Node *) rte->tablesample, 0);
else if (rte->rtekind == RTE_SUBQUERY)
    //子查询有层次关系，因此是1，参照Var的说明
    vars = pull_vars_of_level((Node *) rte->subquery, 1);
else if (rte->rtekind == RTE_FUNCTION)
    vars = pull_vars_of_level((Node *) rte->functions, 0);
else if (rte->rtekind == RTE_TABLEFUNC)
    vars = pull_vars_of_level((Node *) rte->tablefunc, 0);
else if (rte->rtekind == RTE_VALUES)
    vars = pull_vars_of_level((Node *) rte->values_lists, 0);
else
    ……
```

将收集到的需要 Lateral 的 Var 保存到 RelOptInfo->lateral_vars 中就可以了。当然我们还需要注意 PlaceHolderVar 也可能是 Lateral 的，它在对应的 PlaceHolderInfo 中也通过 ph_lateral 变量记录了 Lateral 变量的关系。

4.5.3 收集 Lateral 信息

create_lateral_join_info 函数负责建立表之间的依赖关系，因为 Lateral 变量有“生产者”和“消费者”，生产者需要知道自己要服务于哪些“消费者”，消费者也需要知道自己需要的列属性来自哪个生产者。

在 RelOptInfo 结构体中，有 4 个和 Lateral 相关的变量，我们来看一下这 4 个变量的含义，假如有 SQL 语句：

```
SELECT * FROM STUDENT, LATERAL (SELECT STUDENT.SNO,* FROM SCORE LIMIT 1) SC, LATERAL (SELECT SC.cno,* FROM COURSE LIMIT 1) CO;
```

以这个 SQL 语句为例，SQL 语句中各个表之间的 Lateral 关系如图 4-14 所示。

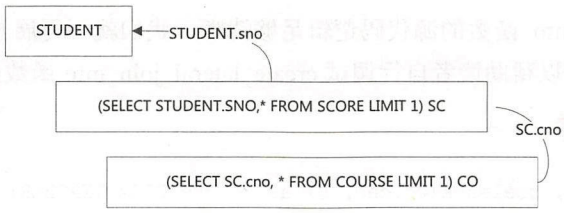


图 4-14　Lateral 依赖关系

direct_lateral_relids：我们能够从语法中直接观察到的 Lateral 变量的从属关系，例如子查询中 SC 引用了 STUDENT 表的列属性 STUDENT.sno，CO 子查询引用了 SC 子查询的列属性 SC.cno。

lateral_relids：在 direct_lateral_relids 的基础上，通过传递闭包推理出的依赖关系，例如子查询 CO 依赖子查询 SC，子查询 SC 依赖表 STUDENT，那么可以获得子查询 CO 依赖表 STUDENT，也就是说子查询 CO 中的 lateral_relids 包含子查询 SC 和表 STUDENT，也就是说 direct_lateral_relids 是 lateral_relids 的子集。

lateral_vars：在 find_lateral_references 函数中收集的 Var 或 PlaceHolderVar。

lateral_referencers：有哪些表引用了“我”的 Var 变量作为 Lateral 变量，它和 lateral_relids 是反向的，记录的是“被依赖”关系。

create_lateral_join_info 函数主要的作用就是填充这些变量的内容，它的主要流程如图 4-15 所示。

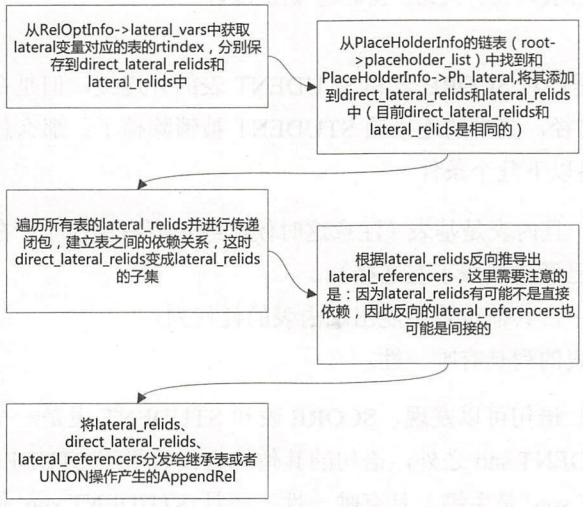


图 4-15　创建并记录 Lateral 关系

create_lateral_join_info 函数的源代码逻辑足够清晰，我们就不再展开论述了，这里给出一个 SQL 语句的例子，可以帮助读者自行调试 create_lateral_join_info 函数的源代码。

```
SELECT * FROM COURSE co
   LEFT JOIN
   LATERAL (SELECT tno, coalesce(co.cno, 0) as co_cno FROM TEACHER) te ON TRUE,
LATERAL (SELECT te.co_cno offset 0) du;
```

create_lateral_join_info 函数是在逻辑优化阶段最后一次对 Lateral 变量进行处理，有了 lateral_relids、lateral_referencers、lateral_vars、direct_lateral_relids 这些变量，在代价优化阶段生成执行路径（Path）的时候，就可以根据这些变量判断表之间的连接顺序、建立参数化路径，这些留到生成执行路径的时候再做分析。

4.6 消除无用连接项

数据库在生成连接路径的时候，需要考虑所有的表之间的连接路径，如果能消除掉其中的一些表，那么势必能够降低物理路径的搜索空间，从而提高查询优化的运行效率，例如有这样一种情况：

```
postgres=# EXPLAIN SELECT SCORE.sno FROM SCORE LEFT JOIN STUDENT ON SCORE.sno = STUDENT.sno;
                QUERY PLAN
---------------------------------------------------
 Seq Scan on score  (cost=0.00..1.01 rows=1 width=4)
(1 row)
```

SQL 语句中原本是针对 SCORE 表和 STUDENT 表的外连接，但是在示例的执行计划中，却显示只有一个扫描路径，其中的连接项 STUDENT 被消除掉了。那么什么样的连接项才能被消除掉呢？它需要满足以下几个条件：

- 必须是左连接，且内表是基表（注意这时候已经没有右连接了，在消除外连接的时候把所有的右连接已经转换成了左连接）。
- 除了当前连接中，其他位置不能出现内表的任何列。
- 连接条件中内表的列具有唯一性。

分析示例中的 SQL 语句可以发现，SCORE 表和 STUDENT 表是一个左连接，除了连接条件 SCORE.sno = STUDENT.sno 之外，语句的其他地方没有引用 STUDENT 表中的任何列，连接条件中的 STUDENT.sno 是主键，具有唯一性，而且 STUDENT.sno 是内表，从执行计划可以看出，STUDENT 表作为无用的连接项，被消除掉了。

```
//1.1 必须是左连接（注意这时候已经没有右连接了，右连接已经转换成了左连接）
if (sjinfo->jointype != JOIN_LEFT ||
   sjinfo->delay_upper_joins)
   return false;

//1.2 且内表是基表
if (!bms_get_singleton_member(sjinfo->min_righthand, &innerrelid))
   return false;

//2.除了当前连接中，其他位置不能出现内表的任何列
joinrelids = bms_union(sjinfo->min_lefthand, sjinfo->min_righthand);
for (attroff = innerrel->max_attr - innerrel->min_attr;
    attroff >= 0;
    attroff--)
{
   if (!bms_is_subset(innerrel->attr_needed[attroff], joinrelids))
      return false;
}

//3.连接条件中内表的列具有唯一性
if (rel_is_distinct_for(root, innerrel, clause_list))
   return true;
```

约束条件在消除无用连接项的过程中会发挥很大的作用，因此在函数 join_is_removable 中会对约束条件进行处理。

首先就可以把 RelOptInfo->joininfo 中的过滤条件先排除，我们已经介绍过，在 RelOptInfo->joininfo 中既有过滤条件也有连接条件，区分的方法是看它们的 RestrictInfo->is_pushed_down 标记，RestrictInfo->is_pushed_down == true 表示这是一个过滤条件，过滤条件不能应用于消除无用连接，因为过滤条件会改变左连接外表的查询结果。另外，即使是连接条件，如果它引用了连接关系之外的表，这个连接条件也不能用于消除无用连接。

其次要求连接条件是 can_join 的和 MergeJoinable 的，也就是说它的形式是 Var = Var 的形式（操作符是否属于 MergeJoinable 可以参考 Form_pg_operator->oprcanmerge）。

再次还要对 Var OP Var 的形式做检查，确定是 Outer_Var OP Inner_Var 还是 Inner_Var OP Outer Var。这样就能保证连接条件符合消除无用连接项的需要，然后还需要对内表的唯一性进行判断，这个过程是在 rel_is_distinct_for 实现的，它把唯一性分成几种情况，我们用示例来说明一下这些情况，并假设有 4 个表分别是：TEST_A(a, b, c, d)、TEST_B(a, b, c, d)、TEST_C(a, b, c, d)、TEST_D(a, b, c, d)。

情况 1：对于普通表（RTE_RELATION）而言，如果表上有主键索引或者 UNIQUE 索引，就可以保证唯一性，这个检查是在 relation_has_unique_index_for 函数中实现的，relation_has_unique_index_for 函数会通过 match_index_to_operand 函数检查连接条件中内表的每一列，例如 TEST_B 表上有索引键值是（a,b,c），那么连接条件中必须全部包含（a,b,c）才能说连接条件符合唯一性。

假如表 TEST_D 上有 UNIQUE 索引 TEST_D_UNI(a, b, c)，那么连接条件如果是 TEST_A.a = TEST_D.a AND TEST_A.b = TEST_D.b 就不满足消除无用连接的条件，因为连接条件中只包含 TEST_D.a 和 TEST_D.b 两个键值，这两个键值不具有唯一性。

```
postgres=# EXPLAIN SELECT TEST_A.a,TEST_A.b FROM TEST_A LEFT JOIN TEST_D ON TEST_A.a = TEST_D.a
AND TEST_A.b = TEST_D.b;
                              QUERY PLAN
-----------------------------------------------------------------------------------------------
 Merge Right Join  (cost=129.05..224.78 rows=1850 width=8)
   Merge Cond: ((test_d.a = test_a.a) AND (test_d.b = test_a.b))
   ->  Index Only Scan using test_d_uni on test_d  (cost=0.15..71.90 rows=1850 width=8)
   ->  Sort  (cost=128.89..133.52 rows=1850 width=8)
         Sort Key: test_a.a, test_a.b
         ->  Seq Scan on test_a  (cost=0.00..28.50 rows=1850 width=8)
(6 rows)
```

而连接条件 TEST_A.a = TEST_D.a AND TEST_A.b = TEST_D.b AND TEST_A.c = TEST_D.c 中包含 UNIQUE 索引的 3 个键值，所以 TEST_D 可以被消除。

```
postgres=# EXPLAIN SELECT TEST_A.a,TEST_A.b FROM TEST_A LEFT JOIN TEST_D ON TEST_A.a = TEST_D.a
AND TEST_A.b = TEST_D.b AND TEST_A.c = TEST_D.c;
                    QUERY PLAN
-----------------------------------------------------------
 Seq Scan on test_a  (cost=0.00..28.50 rows=1850 width=8)
(1 row)
```

情况 2：如果内表是一个子查询，那么如果它的目标里有 DISTINCT 关键字，也能保证唯一性。

```
postgres=# EXPLAIN SELECT TEST_A.a,TEST_A.b FROM TEST_A LEFT JOIN (SELECT DISTINCT A,B FROM
TEST_B) b ON TEST_A.a = b.a AND TEST_A.b = b.b;
                    QUERY PLAN
-----------------------------------------------------------
 Seq Scan on test_a  (cost=0.00..28.50 rows=1850 width=8)
(1 row)
```

情况 3：如果内表是一个子查询，而且子查询有 GROUP BY 子句，那么也能保证唯一性。

```
postgres=# EXPLAIN SELECT TEST_A.a,TEST_A.b FROM TEST_A LEFT JOIN (SELECT A,B FROM TEST_B
GROUP BY a,b) b ON TEST_A.a = b.a AND TEST_A.b = b.b;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on test_a  (cost=0.00..28.50 rows=1850 width=8)
(1 row)
```

情况 4：如果内表是一个子查询，而且子查询的 GROUP BY 子句的键值是空的，这种情况只会返回一行，间接保证了唯一性。

```
postgres=# EXPLAIN SELECT TEST_A.a,TEST_A.b FROM TEST_A LEFT JOIN (SELECT 1 as a, 2 as b FROM
TEST_B GROUP BY ()) b ON TEST_A.a = b.a AND TEST_A.b = b.b;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on test_a  (cost=0.00..28.50 rows=1850 width=8)
(1 row)
```

情况 5：如果内表是一个子查询，而且投影列上有聚集函数，或者在没有 GROUP BY 的情况下，包含 Having 子句，其返回的结果只有一行，间接地保证了唯一性。

```
postgres=# EXPLAIN SELECT TEST_A.a,TEST_A.b FROM TEST_A LEFT JOIN (SELECT MAX(a) as a FROM
TEST_B) b ON TEST_A.a = b.a;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on test_a  (cost=0.00..28.50 rows=1850 width=8)
(1 row)

postgres=# EXPLAIN SELECT TEST_A.a,TEST_A.b FROM TEST_A LEFT JOIN (SELECT SUM(a) as a FROM
TEST_B HAVING SUM(a) > 10) b ON TEST_A.a = b.a;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on test_a  (cost=0.00..28.50 rows=1850 width=8)
(1 row)
```

情况 6：如果内表是一个含有 UNION、EXCEPT、INTERSECT 关键字，但是不带有 ALL 关键字，它的结果也具有唯一性。

```
postgres=# EXPLAIN SELECT TEST_A.a,TEST_A.b FROM TEST_A LEFT JOIN (SELECT a,b FROM TEST_B
UNION SELECT a,b FROM TEST_C) bc ON TEST_A.a = bc.a AND TEST_A.b = bc.
b;
                        QUERY PLAN
```

```
------------------------------------------------------------
 Seq Scan on test_a  (cost=0.00..28.50 rows=1850 width=8)
(1 row)
```

情况 2~6 在 query_is_distinct_for 函数中都有一一对应的源代码，读者可以结合上面的示例对代码进行调试分析，这里不再赘述。

还需要注意到的一种情况，在一个 SQL 语句中可以消除多个表，例如表 TEST_A(a, b)、TEST_B(a, b)、TEST_C(a, b)、TEST_D(a, b)，在 TEST_B、TEST_C、TEST_D 上分别有 TEST_B_PK(a)、TEST_C_PK(a)、TEST_D_PK(a)三个主键索引。

```
postgres=# EXPLAIN SELECT TEST_A.a, TEST_A.b FROM TEST_A LEFT JOIN TEST_B ON TEST_A.a=TEST_B.a
LEFT JOIN TEST_C ON TEST_A.a = TEST_C.a LEFT JOIN TEST_D ON TEST_
A.a = TEST_D.a;
                      QUERY PLAN
------------------------------------------------------------
 Seq Scan on test_a  (cost=0.00..28.50 rows=1850 width=8)
(1 row)
```

这种递进式的删除是通过 remove_useless_joins 函数中的 goto restart 实现的，每当消除掉一个表之后，都重新对所有的左连接再做一次检查，看是否有可能再递进式地消除一个表。

将表从查询树中删除的工作是在 remove_rel_from_query 函数中实现的，它主要做了这么几件事：

- 标记这个表对应的 RelOptInfo 类型为 RELOPT_DEADREL。
- 如果其他表的 RelOptInfo-> attr_needed 中记录了这个表，则将其删除，例如对于连接条件 TEST_A.a = TEST_B.a，需要在 TEST_A.a 对应的 RelOptInfo-> attr_needed 上记录{TEST_A，TEST_B}（在代码中表现为 TEST_A 和 TEST_B 的 Relid），在 TEST_B.a 对应的 RelOptInfo-> attr_needed 上也会记录{TEST_A，TEST_B}（可以参考谓词下推的章节），如果表 TEST_B 可以被消除，那么就需要在 TEST_A.a 对应的 RelOptInfo-> attr_needed 上把 TEST_B 删除。
- 在 SpecialJoinInfo 中还会记录连接的最小集（min_lefthand、min_righthand），如果最小集中包含要消除的表，那么把它清理掉。
- 另外对于 TEST_A.a = TEST_B.a 这样的连接条件，会分别记录到 TEST_A 对应的 RelOptInfo->joininfo 和 TEST_B 对应的 RelOptInfo->joininfo，如果 TEST_B 被消除，那么也需要将连接条件删除掉。这里需要注意不能删除过滤条件，PostgreSQL 数据库的做法是如果 RelOptInfo->joininfo 中的连接条件被删除，过滤条件则在经过清理之后重新调

用 distribute_qual_to_rels 函数进行分发。

4.7　Semi Join 消除

Semi Join 的本质是对于外表也就是 LHS 的每一条元组，如果在内表也就是 RHS 的表中找到一条符合连接条件的元组，就表示连接成功，即使内表中有多个符合连接条件的元组，也只匹配一条。

那么如果内表能保证唯一性，Semi Join 的连接结果就可以和 Inner Join 相同了，因此在这种情况下，可以将 Semi Join 消除，转换为内连接。例如对于具有 DISTINCT 属性的内表，就可以将 Semi Join 转换为 Inner Join。

```
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE a = ANY (SELECT DISTINCT a FROM TEST_B);
                        QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=93.25..145.10 rows=1850 width=16)
   Hash Cond: (test_a.a = test_b.a)
   ->  Seq Scan on test_a  (cost=0.00..28.50 rows=1850 width=16)
   ->  Hash  (cost=70.13..70.13 rows=1850 width=4)
         ->  HashAggregate  (cost=33.13..51.63 rows=1850 width=4)
               Group Key: test_b.a
               ->  Seq Scan on test_b  (cost=0.00..28.50 rows=1850 width=4)
(7 rows)
```

和 remove_useless_joins 函数的情况相似，这里也是通过 rel_is_distinct_for 来判断内表的唯一性的，调用的流程如图 4-16 所示。

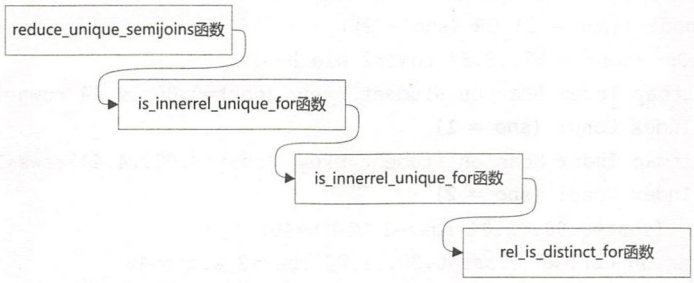


图 4-16　Semi Join 优化流程

4.8 提取新的约束条件

如果有这样一条 SQL 语句：SELECT * FROM STUDENT, COURSE WHERE (sno =1 AND cname='math') OR (sno = 2 AND cname = 'physics')，在对谓词做表达式预处理的时候，我们对这种类型的约束条件无从下手，整个条件作为一个析取范式保存在一个 RestrictInfo 中，这样的约束条件也只能应用到 STUDENT 表和 COURSE 表的连接结果上，对连接操作产生的结果进行过滤。如果有办法把其中的一些条件提取出来，并且能够下推到基表上，那么就会降低连接(join)操作的计算量。

通过分析(sno =1 AND cname='math') OR (sno = 2 AND cname = 'physics')这样一个约束条件就会发现，它是 OR 谓词连接起来的两个合取范式：(sno =1 AND cname='math') 和(sno = 2 AND cname = 'physics')。对于 STUDENT 表而言，能够顺利通过这两个合取范式过滤的元组，一定符合这样一个约束条件：sno = 1 OR sno = 2。同理，对于 COURSE 表而言，也能够得到一个类似的约束条件：cname = 'math' OR cname = 'physics'。如果先把这两个约束条件应用到基表（STUDENT 表和 COURSE 表）上，显然就能实现我们降低计算量的目标。PostgreSQL 数据库通过 extract_restriction_or_clauses 函数实现了这样的功能，我们来看一下 SQL 语句的执行计划。

```
postgres=# EXPLAIN SELECT * FROM STUDENT, COURSE WHERE (sno=1 AND cname='math') OR (sno=2
AND cname='physics');
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------
-----------------------------------------
 Nested Loop  (cost=8.87..17.99 rows=3 width=65)
   Join Filter: (((student.sno = 1) AND ((course.cname)::text = 'math'::text)) OR
((student.sno = 2) AND ((course.cname)::text = 'physics'::text)))
   -> Bitmap Heap Scan on student  (cost=8.87..16.86 rows=2 width=19)
        Recheck Cond: ((sno = 1) OR (sno = 2))
        -> BitmapOr  (cost=8.87..8.87 rows=2 width=0)
             -> Bitmap Index Scan on student_pkey  (cost=0.00..4.44 rows=1 width=0)
                  Index Cond: (sno = 1)
             -> Bitmap Index Scan on student_pkey  (cost=0.00..4.44 rows=1 width=0)
                  Index Cond: (sno = 2)
   -> Materialize  (cost=0.00..1.04 rows=2 width=46)
        -> Seq Scan on course  (cost=0.00..1.03 rows=2 width=46)
             Filter: (((cname)::text = 'math'::text) OR ((cname)::text = 'physics'::text))
(12 rows)
```

从执行计划可以看出 sno = 1 OR sno = 2 和 cname = 'math' OR cname = 'physics'都应用到了

基表上，形成了过滤条件，但是这时候约束条件(sno=1 AND cname='math') OR (sno=2 AND cname='physics')还是存在的，因此最终形成的约束条件是((sno=1 AND cname='math') OR (sno=2 AND cname='physics')) AND (sno = 1 OR sno = 2) AND (cname = 'math' OR cname = 'physics')。

4.8.1　提取需要满足的条件

由于我们已经默认约束条件的顶层是一个合取范式，合取范式的下一层子句可能就是析取范式，这些析取范式如果是基于 OR 谓词的，就有可能提取出新的约束条件。例如对于约束条件((sno=1 AND cname='math') OR (sno=2 AND cname='physics')) AND sname='JIM'，显然这个约束条件是析取范式((sno=1 AND cname='math') OR (sno=2 AND cname='physics'))和单个子句 sname='JIM'形成的一个合取范式，其中析取范式((sno=1 AND cname='math') OR (sno=2 AND cname='physics'))是 OR 谓词连接的，就有可能提取出新的约束条件。

提取新的约束条件还需要满足析取范式中的每个子句（也就是 OR 谓词连接起来的每个下层的子约束条件）都同时引用了同一个表，也就是说(sno=1 AND cname='math')中被引用的 sno 是 STUDENT 表的属性，(sno=2 AND cname='physics')中被引用的 sno 也是 STUDENT 表的属性，约束条件 sno = 1 OR sno = 2 才能被提取出来，否则就无法提取出新的约束条件。例如对于((sno=1 AND cname='math') OR (sno=2 AND cname='physics') OR sname = 'TOM'，由于 sname = 'TOM'的存在，就无法提取出新的约束条件。

4.8.1.1　join_clause_is_movable_to 函数

提取新的约束条件的时候还需要满足谓词下推的规则，这样新生成的约束条件才可以认为是过滤条件而下推给基表，PostgreSQL 数据库通过 join_clause_is_movable_to 函数来判断这种情况。例如对于一个左连接 SQL 语句，即使我们可以生成一个约束条件 sno = 1 OR sno = 2，但是这时它是一个连接条件，而属性 sno 所在的 STUDENT 表是左连接的 Nonnullable-side，这样的连接条件不能下推给基表，因此生成这样的连接条件是没用的，从下面的示例中也可以看出并没有生成 sno = 1 OR sno = 2 这样的连接条件。

```
postgres=# EXPLAIN SELECT * FROM STUDENT LEFT JOIN COURSE ON (sno=1 AND cname='math') OR (sno=2
AND cname='physics');
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------
-------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..119286.58 rows=1289119 width=65)
   Join Filter: (((student.sno = 1) AND ((course.cname)::text = 'math'::text)) OR
((student.sno = 2) AND ((course.cname)::text = 'physics'::text)))
```


非卖品，仅供非商业用途或交流学习使用

```
  ->  Seq Scan on student  (cost=0.00..61275.19 rows=1289119 width=19)
  ->  Materialize  (cost=0.00..1.04 rows=2 width=46)
        ->  Seq Scan on course  (cost=0.00..1.03 rows=2 width=46)
              Filter: (((cname)::text = 'math'::text) OR ((cname)::text = 'physics'::text))
(6 rows)
```

如果约束条件中引用了下层的子外连接的 Nullable-side 的表，那么这种约束条件也不能应用于提取新的约束条件，例如 SQL 语句 EXPLAIN SELECT * FROM SCORE LEFT JOIN (STUDENT FULL JOIN COURSE ON TRUE) sc ON (SCORE.sno = 1 AND sc.sno = 1) OR (SCORE.sno = 2 AND sc.sno = 2) AND sc.sname IS NULL AND sc.cname IS NULL。

如果约束条件中引用了 LATERAL 的表，那么这种约束条件是不能用于提取新的约束条件的，例如 SQL 语句 SELECT * FROM STUDENT, LATERAL(SELECT sno,cno FROM SCORE WHERE STUDENT.sno = SCORE.sno GROUP BY SCORE.cno,SCORE.sno) sc WHERE (sc.cno = 1 AND STUDENT.sname = 'JACK') OR (sc.cno = 2 AND STUDENT.sname = 'TOM')，读者可以根据上面两个示例自行调试代码加深理解。

除了 join_clause_is_movable_to 函数之外，在提取的过程中，还会通过 is_safe_restriction_clause_for 函数来判断如下情况：子约束条件中不含有易失性函数、子约束条件不是常量约束条件、子约束条件中只引用了基表一个表，否则也不能提取新的约束条件。

```
//常量约束条件
if (rinfo->pseudoconstant)
   return false;

//约束条件中只包含一个基表
if (!bms_equal(rinfo->clause_relids, rel->relids))
   return false;

//约束条件中不含有易失性函数
if (contain_volatile_functions((Node *) rinfo->clause))
   return false;
```

4.8.2 提取流程

提取新约束条件的过程在 extract_or_clause 函数中实现，它的参数主要是一个基于 OR 谓词的析取范式和要提取哪个基表的约束条件，主要提取流程是遍历 OR 谓词连接的子约束条件，下面我们来分析一下它的源代码。

```
//对于约束条件(Pa1 AND Pb2) OR (Pa2 AND Pb2)，分两次遍历
```


非卖品，仅供非商业用途或交流学习使用

```
//首先查看(Pa1 AND Pb2)，然后查看(Pa2 AND Pb2)
foreach(lc, ((BoolExpr *) or_rinfo->orclause)->args)
{
    ......
    //如果是(Pa1 AND Pb2)这种形式
    if (and_clause(orarg))
    {
        ......
        foreach(lc2, andargs)
        {
            RestrictInfo *rinfo = lfirst_node(RestrictInfo, lc2);
            if (restriction_is_or_clause(rinfo))
            {
                //递归调用 extract_or_clause 函数
                //对下层的 OR 谓词连接的析取范式尝试提取约束条件
                suborclause = extract_or_clause(rinfo, rel);
                if (suborclause)
                    subclauses = lappend(subclauses, suborclause);
            }
            else if (is_safe_restriction_clause_for(rinfo, rel))
                subclauses = lappend(subclauses, rinfo->clause);
        }
    }
    //也有可能是一个单独的子句，例如对于(Pa1 AND Pb2) OR (Pa2 AND Pb2) OR Pa3
    //其中 Pa3 就是一个单独的子句
    else
    {
        ......
        if (is_safe_restriction_clause_for(rinfo, rel))
            subclauses = lappend(subclauses, rinfo->clause);
    }

    //如果一个子约束条件中没提取到对应基表的约束条件，那么终止整个提取过程
    if (subclauses == NIL)
        return NULL;

    //记录提取的约束条件
    subclause = (Node *) make_ands_explicit(subclauses);
    if (or_clause(subclause))//如果提取的约束条件是 OR 谓词连接的，尝试“拉平”
        clauselist = list_concat(clauselist,
                                 list_copy(((BoolExpr *) subclause)->args));
```


非卖品，仅供非商业用途或交流学习使用

```
    else
        clauselist = lappend(clauselist, subclause);
}
```

4.8.3 选择率修正

这里我们要讨论一些“超纲”的知识，因为我们还没有完整地介绍过选择率，读者可以先去查阅选择率的相关章节，然后再返回来看这部分内容。

由于约束条件的改变，选择率的估算就会发生变化，但这种变化显然是没有“道理”的，因为我们新添加的约束条件是从原来的约束条件中衍生出来的。例如对于约束条件((sno=1 AND cname='math') OR (sno=2 AND cname='physics'))，它本来是应用在 STUDENT 表和 COURSE 表的连接结果之上的，对于这样一个约束条件的选择率是 P((sno=1 AND cname='math') OR (sno=2 AND cname='physics'))。

这个选择率的计算都是基于 STUDENT 表的统计信息和 COURSE 表的统计信息进行的，而这些统计信息又是基于整个表的数据经过采样和统计分析获得的，也就是说上面的选择率计算是基于 STUDENT 表和 COURSE 表的全体数据进行的。

假如这时候提取出新的约束条件 sno = 1 OR sno = 2，那么这个约束条件就会先对 STUDENT 表进行一次过滤，过滤产生的结果用于和 COURSE 表做连接，那么 P((sno=1 AND cname='math') OR (sno=2 AND cname='physics'))中基于 STUDENT 全部数据计算的选择率就不再准确了（因为过滤导致 STUDENT 表变小了），这时候就需要调整 P((sno=1 AND cname='math') OR (sno=2 AND cname='physics'))的选择率。

假设基表 STUDENT 中有 m 条数据，基表 COURSE 中有 n 条数据，那么在没有增加新的约束条件时，估算最终产生的结果数量为：

$$m \times n \times \text{P((sno=1 AND cname='math') OR (sno=2 AND cname='physics'))}$$

STUDENT 表经过约束条件 sno = 1 OR sno = 2 过滤后产生的结果数量为：

$$m \times \text{P(sno = 1 OR sno = 2)}$$

如果我们不对选择率进行修正，新加约束条件之后，连接产生的结果数量为：

$$m \times \text{P(sno = 1 OR sno = 2)} \times n \times \text{P((sno=1 AND cname='math') OR (sno=2 AND cname='physics'))}$$

那么我们可以在新增加约束条件之后做如下修正：

非卖品，仅供非商业用途或交流学习使用

$$m \times P(sno = 1 \text{ OR } sno = 2) \times n \times [P((sno=1 \text{ AND } cname='math') \text{ OR } (sno=2 \text{ AND } cname='physics')) / P(sno = 1 \text{ OR } sno = 2)]$$

也就是说，在增加新的约束条件之后，约束条件((sno=1 AND cname='math') OR (sno=2 AND cname='physics'))的选择率应该修正为：

$$P((sno=1 \text{ AND } cname='math') \text{ OR } (sno=2 \text{ AND } cname='physics')) / P(sno = 1 \text{ OR } sno = 2)$$

```
//新提取的约束条件的选择率
or_selec = clause_selectivity(root, (Node *) or_rinfo, 0, JOIN_INNER, NULL);

//如果新提取的约束条件的选择率比较高，就证明过滤掉的元组比较少，忽略这个约束条件
if (or_selec > 0.9)
   return;                /* forget it */

if (or_selec > 0)
{
   //计算原始的约束条件的选择率
   orig_selec = clause_selectivity(root, (Node*)join_or_rinfo, 0, JOIN_INNER, &sjinfo);

   //对选择率做修正
   join_or_rinfo->norm_selec = orig_selec / or_selec;

   //校正选择率
   if (join_or_rinfo->norm_selec > 1)
      join_or_rinfo->norm_selec = 1;
}
```

4.9　小结

本章的内容是查询优化中的一个难点，由于有 LeftJoin、SemiJoin、AntiJoin 等形式的连接操作，导致谓词下推和连接顺序交换的难度增大。本书虽然给出了一些结论和规则，但是除此之外，读者要想掌握这部分内容还需要准确理解 LeftJoin、SemiJoin、AntiJoin 的具体含义，这是谓词下推和连接顺序交换的基础。另外还需要大量地阅读执行计划，做到可以根据 SQL 语句手写物理执行计划。

PostgreSQL 数据库目前只做到了基于等价约束条件的等价推理，实际上一些商业数据库已经做到可以根据非等价的约束条件进行基于范围的推理，这部分是 PostgreSQL 数据库可以改进的内容。

非卖品，仅供非商业用途或交流学习使用

5 第5章 统计信息和选择率

PostgreSQL 数据库的物理优化需要计算各种物理路径的代价，而代价估算的过程严重地依赖于数据库的统计信息，统计信息是否能准确地描述表中的数据分布情况是决定代价的准确性的重要条件之一。

通过统计信息，代价估算系统就可以了解一个表有多少行数据、用了多少个数据页面、某个值出现的频率等，然后根据这些信息计算出一个约束条件能过滤掉多少数据，这种约束条件过滤出的数据占总数据量的比例称为“选择率”。

$$选择率 = \frac{约束条件过滤后的元组数}{约束条件过滤前的总元组数}$$

古人云“兵马未动，粮草先行”，统计信息和选择率都是代价估算过程中的“粮草”，在开始物理优化的源代码分析之前，我们先来分析一下统计信息的获取过程和选择率的计算过程。

5.1 统计信息

PostgreSQL 数据库支持多种形式的统计信息，包括单列的统计信息和多列（扩展）的统计

非卖品，仅供非商业用途或交流学习使用

信息，单列的统计信息是指对每个表的每一个属性（列）都在 PG_STATISTIC 系统表中产生一个对应的统计信息元组（行），这个元组负责从多个角度描绘这个属性中的数据分布，单列统计信息主要包含表 5-1 中的这些形式。

表 5-1　单列统计信息类型说明

类型	说明
STATISTIC_KIND_MCV	高频值（常见值），在一个列里出现最频繁的值，按照出现的频率进行排序，并且生成一个一一对应的频率数组，这样就能知道一个列中有哪些高频值，这些高频值的频率是多少
STATISTIC_KIND_HISTOGRAM	直方图，PostgreSQL数据库使用等频直方图来描述一个列中的数据的分布，高频值不会出现在直方图中，这就保证了数据的分布是相对平坦的
STATISTIC_KIND_CORRELATION	相关系数，相关系数记录的是当前列未排序的数据分布和排序后的数据分布的相关性，这个值通常在索引扫描时用来估计代价，假设一个列未排序和排序之后的相关性是0，也就是完全不相关，那么索引扫描的代价就会高一些
STATISTIC_KIND_MCELEM	类型高频值（常见值），用于数组类型或者一些其他类型，PostgreSQL数据库提供了ts_typanalyze系统函数来负责生成这种类型的统计信息
STATISTIC_KIND_DECHIST	数组类型直方图，用于给数组类型生成直方图，PostgreSQL数据库提供了array_typanalyze系统函数来负责生成这种类型的统计信息
STATISTIC_KIND_RANGE_LENGTH_HISTOGRAM	为Range类型生成一个基于长度的直方图统计信息，用户可以自定义Range类型，例如CREATE TYPE floatrange AS RANGE (subtype = float8, subtype_diff = float8mi)，PostgreSQL数据库提供了range_typanalyze系统函数负责生成这种类型的统计信息
STATISTIC_KIND_BOUNDS_HISTOGRAM	为Range类型生成一个基于边界的直方图，这种类型的直方图也通过range_typanalyze系统函数来进行统计

STATISTIC_KIND_MCV、STATISTIC_KIND_HISTOGRAM、STATISTIC_KIND_CORRELATION 是统计模块常用的 3 种统计方式，其他统计方式和这 3 种统计方式大致上类似，但是针对的是一些特殊的类型。

使用基于单列的统计信息来对基于单个列的约束条件（如 a = 1 这种约束条件）进行选择率的估计，误差范围是可控的，但是对于引用了多个列的约束条件（如 a = 1 OR b = 2 AND c = 3 这种约束条件），如果还使用单列的统计信息进行估算，就需要将这个约束条件拆分成多个独

非卖品，仅供非商业用途或交流学习使用

立的子约束条件，对每个子约束条件进行选择率估算，并且假设这些子约束条件的选择率是独立的（即假设 a = 1、b=2、c=3 分别独立），然后基于概率的方法对总的选择率进行估算，由于在实际应用中并不能保证它们是独立的，因此可能导致估算的误差较大，PostgreSQL 数据库对统计信息的功能进行了扩展，支持了多列的统计信息用来计算各个列之间的依赖度，它主要包括两种形式，如表 5-2 所示。

表 5-2　多列统计信息类型说明

类型	说明
STATS_EXT_NDISTINCT	和单列统计信息中的stadistinct是类似的，stadistinct中记录的是单列中去掉NULL值和消重之后的数据量或者比例，STATS_EXT_NDISTINCT类型的统计信息则记录的是基于多列的消重之后的数据量
STATS_EXT_DEPENDENCIES	计算各个列之间的函数依赖度，通过函数依赖度计算各个列之间的依赖关系，从而得到准确的统计信息

获得了统计信息之后，在代价估算的时候就可以利用这些统计信息进行计算，比如可以借用统计信息计算约束条件的选择率：

```
--STUDENT 表中需要多些数据
DELETE FROM STUDENT;
INSERT INTO STUDENT SELECT GENERATE_SERIES(1,10000), LEFT(RANDOM()::TEXT, 10), 1;
ANALYZE STUDENT;

--选择率高，采用顺序扫描的方法获取数据
postgres=# EXPLAIN SELECT * FROM STUDENT WHERE sno > 2;
                      QUERY PLAN
-------------------------------------------------------------
 Seq Scan on student  (cost=0.00..224.00 rows=9999 width=10)
   Filter: (sno > 2)
(2 rows)

--选择率低，采用索引扫描的方法获取数据
postgres=# EXPLAIN SELECT * FROM STUDENT WHERE sno < 2;
                            QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using student_pkey on student  (cost=0.29..8.30 rows=1 width=10)
   Index Cond: (sno < 2)
(2 rows)
```


非卖品，仅供非商业用途或交流学习使用

通过示例可以看出，在选择率高的情况下查询路径选择了顺序扫描（SeqScan）的方式，在选择率低的情况下查询路径选择了索引扫描（IndexScan）。这是因为对于索引扫描而言，它产生了“随机读”，如图 5-1 所示，PostgreSQL 数据库中堆表的数据是无序的，而主键索引则以 B 树的方式进行存储，B 树的叶子节点是有序的，如果通过顺序扫描（SeqScan）的方式对 STUDENT 表进行遍历，则需要对每一个 STUDENT 表中的元组应用约束条件，筛选出符合约束条件的元组作为结果。如果通过索引扫描的方式，则可以借助 B 树索引的有序性，快速定位索引项的位置，每一个索引项都有一个堆元组的“地址”，这个“地址”指向了该索引项对应的 STUDENT 表中的元组，因为索引项是有序的，而 STUDENT 表中的元组是无序的，所以这时就产生了随机读。

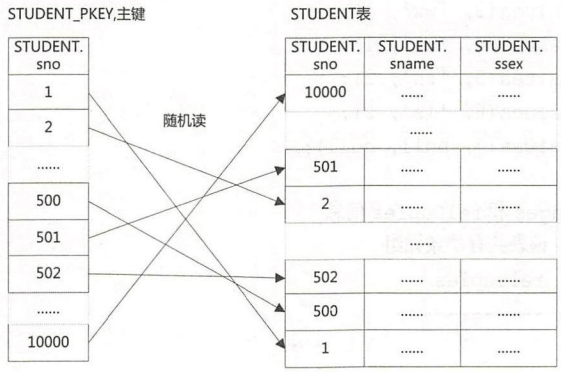


图 5-1　索引会产生随机读的问题

而在 PostgreSQL 数据库中对于顺序读和随机读定义了不同的代价。

```
#define DEFAULT_SEQ_PAGE_COST  1.0      //顺序读的单页代价
#define DEFAULT_RANDOM_PAGE_COST  4.0  //随机读的单页代价
```

如果选择率比较高，那么随机读的代价累计起来就很可观了，因此在选择率高的情况下会选择顺序扫描，而当选择率比较低时，顺序扫描仍然要把整个表的数据过滤一遍。索引扫描的单个随机读代价虽然高，但总量远远小于顺序读的数据量，因此顺序读的累计的代价就会超过索引扫描的代价，这时就会选择索引扫描作为执行路径。

5.1.1　PG_STATISTIC 系统表

PostgreSQL 数据库在 PG_CLASS 系统表中会保存两个统计信息，分别是 relpages 和 reltuples，relpages 记录了当前表占用了多少个页面，reltuples 记录了当前表共有多少条元组。另外，PostgreSQL 数据库使用 PG_STATISTIC 系统表保存单列的统计信息，如果用户要给某一个表生

非卖品，仅供非商业用途或交流学习使用

成统计信息，则可以使用 ANALYZE 语句对一个表进行统计分析，这样就能给这个表生成统计信息并且保存在 PG_STATISTIC 系统表中。

下面主要看一下 PG_STATISTIC 系统表的统计信息的组织形式。以 STUDENT 表为例，我们向 STUDENT 表插入一些数据，然后对 STUDENT 表执行 ANALYZE 操作，由于 STUDENT 表有 3 个列属性，因此执行 ANALYZE 操作之后会在 PG_STATISTIC 系统表中给 STUDENT 表生成 3 行统计信息。

```
--向 STUDENT 表插入一些数据
insert into student values(1, 'zs', 1);
insert into student values(2, 'ls', 1);
insert into student values(3, 'ww', 1);
insert into student values(4, 'zl', 1);
insert into student values(5, 'zs', 2);
insert into student values(6, 'ls', 2);
insert into student values(7, null, null);

--查询 STUDENT 表的 relpages 和 reltuples 信息
--可以看出占用了 1 个页面，该表共有 7 条元组
 relname  | relpages | reltuples
----------+----------+-----------
 student  |        1 |         7
(1 row)

--对 STUDENT 表做统计
ANALYZE STUDENT;

--查询 PG_STATISTIC 表中和 STUDENT 表相关的统计信息，其中 24582 是 STUDENT 表的 OID
--staattnum 中的 1、2、3 分别对应 STUDENT 表中的 3 个列
starelid |staattnum | stanullfrac | stawidth | stadistinct
---------+---------- +-------------+----------+-------------
   24582 |        1 |           0 |        4 |          -1
   24582 |        2 |    0.142857 |        3 |   -0.571429
   24582 |        3 |    0.142857 |        4 |   -0.285714
(3 rows)
```

starelid/staattnum：对应的表的 OID（来自 PG_CLASS）和列的编号（来自 PG_ATTRIBUTES），示例中 STUDENT 表的 OID 为 24582，其中 sno、sname、ssex 三个属性的编号分别是 1、2、3。

stanullfrac：NULL 值率，表示一个属性（列）里 NULL 值所占的比例，如示例所示，STUDENT

表的 sname 和 ssex 两个列中各有一个 NULL 值，因此它们 NULL 值率都是 1/7 = 0.142857。

stawidth：计算该属性（列）的平均宽度，STUDENT 表的第一个属性 sno 是 INT 类型，INT 类型是定长类型，它的宽度是 4，而 STUDENT 表的第二个属性 sname 是变长的类型，它的宽度是所有值的平均宽度，虽然 sname 的类型是 VARCHAR(10)，但是实际上我们插入的值全部长度都是 2，再加上 VARCHAR 类型的 Header 的 1 个字节的长度，可以获得它的平均宽度是 3。

stadistinct：计算该属性消重之后的数据的个数或比例，stadistinct 的取值有 3 种情况。

- = 0，代表未知或者未计算的情况。
- > 0，代表消除重复值之后的个数，不常使用这种情况。
- < 0，其绝对值是去重之后的个数占总个数的比例，通常使用这种类型。

STUDENT 表的第二列共有 7 个值，去掉 NULL 值，去掉重复值后还剩 4 个值（zs, ls, ww, zl），因此 stadistinct = 4/7 = 0.571429。

PostgreSQL 数据库对每一个属性（列）的统计目前最多只能应用 STATISTIC_NUM_SLOTS = 5 种方法，因此在 PG_STATISTIC 中会有 stakind(1-5)、staop(1-5)、stanumbers[1](1-5)、stavalues(1-5)，分别是 5 个槽位。如果 stakind 不为 0，那么表明这个槽位有统计信息，第一个统计方法统计的信息首先会记录到第一个槽位（stakind1、staop1、stanumbers1[1]、stavalues1），第二个统计方法的统计信息会记录到第二个槽位（stakind2、staop2、stanumbers2[1]、stavalues2），依此类推。

stakind：统计信息的的形式，它的定义如下：

```
#define STATISTIC_KIND_MCV          1
#define STATISTIC_KIND_HISTOGRAM      2
#define STATISTIC_KIND_CORRELATION      3
#define STATISTIC_KIND_MCELEM         4
#define STATISTIC_KIND_DECHIST         5
#define STATISTIC_KIND_RANGE_LENGTH_HISTOGRAM   6
#define STATISTIC_KIND_BOUNDS_HISTOGRAM        7
```

在对 STUDENT 表做完统计后，STUDENT 的每个行的槽位情况如下：

```
--查询 STUDENT 表中的每个属性所应用的方法
--STUDENT(sno)应用了 2 个方法，占 2 个槽位，分别是直方图（2）和相关系数（3）
--STUDENT(sname)应用了 3 个方法，占 3 个槽位，分别是常见值（1）、直方图（2）和相关系数（3）
--STUDENT(ssex)应用了 2 个方法，占 2 个槽位，分别是常见值（2）和相关系数（3）
starelid | staattnum | stakind1 | stakind2 | stakind3 | stakind4 | stakind5
---------+-----------+----------+----------+----------+----------+---------
```

```
  24582 |       1 |      2 |      3 |      0 |      0 |      0
  24582 |       2 |      1 |      2 |      3 |      0 |      0
  24582 |       3 |      1 |      3 |      0 |      0 |      0
(3 rows)
```

staop：统计过程中涉及的操作符。

stanumbers：存放统计信息的高频值数组或者存放相关系数，例如 stakind1 保存的统计信息类型是 STATISTIC_KIND_MCV，那么在 stanumbers1 中保存的就是高频值数组，数组中记录的是每个高频值所占的频率值，再例如 stakind3 中保存的统计信息类型是 STATISTIC_KIND_CORRELATION，那么在 stanumbers3 中保存的就是相关系数。

stavalues：统计值的数组，如果 stakind1 保存的统计信息类型是 STATISTIC_KIND_MCV，那么在 stavalues 中保存的就是高频值对应的值，它和 stanumbers 中的高频值数组的频率值一一对应。如果 stakind1 保存的统计信息类型是 STATISTIC_KIND_HISTOGRAM，那么在 stanumbers1 中保存的是直方图中的桶的信息，由于直方图是等频直方图，因此只要记录了每个桶的边界值，就可以获得每个桶的平均比例。

通过示例可以看出，不同的列使用的槽数是不同的，例如 STUDENT.sno 列和 STUDENT.ssex 列都使用了 2 个槽，而 STUDENT.sname 列则使用了 3 个槽。

下面可以通过查看每个槽的信息，来分析一下 STUDENT 表的数据分布情况。

STUDENT.sno 属性在第一个槽中保存的是直方图信息，因为 STUDENT.sno 上面有主键索引，因此它的直方图的每个桶的边界值都是平坦的。

STUDENT.sname 属性和 STUDENT.ssex 属性在第一个槽中保存的是 STATISTIC_KIND_MCV 形式的统计信息，以 STUDENT.sname 为例，{ls, zs}表示在 STUDENT.sname 中这两个值出现的频率比较高，{0.285714, 0.285714}和{ls, zs}一一对应，表示的是每个高频值的频率。

```
--第一个槽的统计信息
starelid | staattnum| stakind1 | staop1 |    stanumbers1     |  stavalues1
---------+----------+----------+--------+--------------------+-------------
  24582 |       1  |      2   |    97  |                    | {1,2,3,4,5,6,7}
  24582 |       2  |      1   |    98  |{0.285714,0.285714} | {ls,zs}
  24582 |       3  |      1   |    96  |{0.571429,0.285714} | {1,2}
(3 rows)
```

STUDENT.sname 属性在第二个槽中保存的是直方图信息，STUDENT.sno 和 STUDENT.ssex

属性在第二个槽中保存的是相关系数，从示例可以看出它们的相关系数都是 1，也就是说数据的分布和排序后的数据分布完全正相关，这是因为 STUDENT 表中的数据写入后没有进行过更新操作，目前在堆中保存的顺序和插入数据时的顺序是一致的，而插入数据时这两个列的数据是有序的。

```
--第二个槽的统计信息
starelid | staattnum | stakind2 | staop2 | stanumbers2 | stavalues2
---------+-----------+----------+--------+-------------+------------
   24582 |         1 |        3 |     97 | {1}         |
   24582 |         2 |        2 |    664 |             | {ww,zl}
   24582 |         3 |        3 |     97 | {1}         |
(3 rows)
```

STUDENT.sname 属性在第三个槽中保存的是相关系数信息，可以看出 STUDENT.sname 中的堆数据的顺序和排序后的顺序的相关系数是 0.0285714。

```
--第三个槽
starelid  | staattnum | stakind3 | staop3 | stanumbers3 | stavalues3
---------+-----------+----------+--------+-------------+------------
   24582 |         1 |        0 |      0 |             |
   24582 |         2 |        3 |    664 | {0.0285714} |
   24582 |         3 |        0 |      0 |             |
(3 rows)
```

5.1.2　PG_STATISTIC_EXT 系统表

PG_STATISTIC_EXT 系统表保存的是多列的统计信息，用户需要显式地使用 CREATE STATISTICS 语句对一个表创建多列统计信息，例如：

```
postgres=# CREATE STATISTICS STATEXT_STUDENT ON sno, sname, ssex FROM STUDENT;
CREATE STATISTICS
postgres=# SELECT * FROM PG_STATISTIC_EXT WHERE stxname='statext_student';
 stxrelid|    stxname     | stxnamespace | stxowner | stxkeys | stxkind | stxndistinct | stxdependencies
--------+---------------+--------------+----------+---------+---------+--------------+---------------
   16384 |statext_student |         2200 |       10 | 1 2 3   | {d,f}   |              |
(1 row)
```

通过 CREATE STATISTICS 语句创建统计信息之后只是在 PG_STATISTIC_EXT 系统表中增加了一个统计信息项，这时候并没有真正对指定表上的属性去做统计分析，只有在用户对表再次执行 ANALYZE 的时候，而且 ANALYZE 的表的属性满足了多列统计信息的要求，才会生成多列统计信息。

```
-- ANALYZE 语句中指定的属性和多列统计信息项 STATEXT_STUDENT 中要求的不一致
postgres=# ANALYZE STUDENT(sno, sname);
WARNING:  statistics object "public.statext_student" could not be computed for relation
"public.student"
ANALYZE
-- ANALYZE 语句中指定的属性和多列统计信息项 STATEXT_STUDENT 中要求的一致
postgres=# ANALYZE STUDENT(sno, sname, ssex);
ANALYZE

postgres=# SELECT * FROM PG_STATISTIC_EXT WHERE stxname='statext_student';
 stxrelid |    stxname     | stxnamespace | stxowner | stxkeys | stxkind |
stxndistinct                |
                                        stxdependencies
----------+----------------+--------------+----------+---------+---------+-------------
-----------------------------------+-----------------------------------
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------
   16384 | statext_student |         2200 |       10 | 1 2 3   | {d,f}   | {"1, 2": 7, "1, 3":
7, "2, 3": 7, "1, 2, 3": 7} | {"1 => 2": 1.000000, "1 => 3": 1.0
00000, "2 => 1": 0.428571, "2 => 3": 0.428571, "3 => 1": 0.142857, "3 => 2": 0.142857, "1,
2 => 3": 1.000000, "1, 3 => 2": 1.000000, "2, 3 => 1": 1.000000}
(1 row)
```

PG_STATISTIC_EXT 系统表中的每个属性的说明如表 5-3 所示。

表 5-3　多列统计信息存储的系统表

属性	类型	说明
stxrelid	Oid	统计信息属于哪个表
stxname	NameData	统计信息的名字
stxnamespace	Oid	统计信息的namespace
stxowner	Oid	统计信息的创建者
stxkeys	int2vector	统计哪些列
stxkind	char	多列统计的类型，目前支持两种类型： #define STATS_EXT_NDISTINCT 'd' #define STATS_EXT_DEPENDENCIES 'f'
stxndistinct	pg_ndistinct	STATS_EXT_NDISTINCT类型的统计信息
stxdependencies	pg_dependencies	STATS_EXT_DEPENDENCIES类型的统计信息

版权所有，严禁传播，违者自负法律责任！

5.1.3　单列统计信息生成

统计信息的生成主要在 analyze_rel 函数->do_analyze_rel 函数中，统计信息的生成过程主要分成两个部分：数据采样和数据统计，如图 5-2 所示。

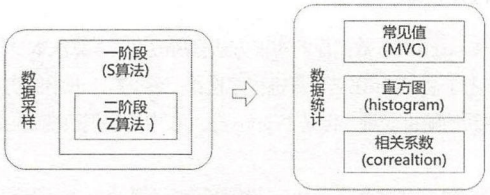


图 5-2　单列统计信息生成的步骤

5.1.3.1　采样

PostgreSQL 数据库的统计模块对一个表进行统计的时候，不是使用全部数据作为样本进行统计，而是随机地采集表中的一部分元组作为样本来生成统计信息。通过调整 default_statistics_target 的值可以改变样本容量，目前 default_statistics_target 的默认值是 100，在计算样本容量时，采用 300×default_statistics_target=30000 作为采样的样本默认容量。

根据 SIGMOD98 中的论文 *Random sampling for histogram construction: how much is enough*，这个样本容量可以满足采样的显著性，但是当前社会是一个数据爆炸的时代，数据库需要存储的数据量越来越大，30000 的样本容量是否能显著地代表整个表中的数据分布就值得商榷了。假如一个表的数据量小于 30000，那么所有的数据全部纳入样本空间，它的采样率达到 100%，基于这个样本空间的统计结果肯定是准确的，而假如有一个 30000000 条数据的表，采样率只有 30000/30000000 = 1‰，这样的样本空间就可能产生误差，而随着数据的增多，样本空间所占的比例也越来越小，因此 PostgreSQL 数据库允许我们通过调节 default_statistics_target 来调整采样容量。还需要注意的是，在生成统计信息的过程中需要对样本进行多次排序（qarg_sort），增大样本空间会降低统计信息计算的性能，因此需要注意在采样的显著性和统计信息的计算性能之间做平衡。

目前 PostgreSQL 数据库的数据采样的实现代码在 acquire_sample_rows 函数中，它采用了两阶段采样的算法进行数据的采样，第一个阶段使用 S 算法对表中的页面进行随机采样，第二阶段使用 Z(Vitter)算法，它在第一阶段采样出来的页面的基础上对元组进行采样。

数据采样的两个阶段采用不同的算法是因为当对一个表进行统计分析的时候，它的页面数（块数）是可以准确获得到的，也就是说页面采样是在已知总体容量的基础上进行的。而第二阶段的 Z（Vitter）算法是一种蓄水池算法，它主要解决的是在不知道总体容量的情况下如何进

版权所有，严禁传播，违者自负法律责任！

行随机采样，第一阶段数据采样产生了一组页面，第二阶段数据采样会在第一阶段采样产生的页面上对元组进行采样，但在对页面进分析之前，无法知道一个页面上有多少个元组，因此也就无法知道第二阶段数据采样的总体容量，因此需要采用蓄水池技术，先将蓄水池“蓄满水”，然后在随机替换蓄水池中的元组。

S 算法（选择抽样技术），PostgreSQL 数据库的实现方法有所改进，S 算法本身需要多次获取随机数，这里简化成只获取一次随机数，这里的实现参考了 Knuth 在《计算机程序设计，卷 2》一书中的实现，在实现时也结合了数据库的本身情况，例如原算法每次都会产生新的随机变量，但是 PostgreSQL 中的实现改变为 S6(p*=1-k/K)，这样减小 p 的值以满足 S4 的条件。

从 N 个记录中随机地选取 n 个记录，其中 0<n<=N，[设置初值]设置 t = 0（表示已经选出的块数），m = 0（表示已经处理过的块数），下面描述 PostgreSQL 中关于改进型 S 算法的实现。

S1. [设置变量] 设置 K = N - t(尚未处理的块数)，k = n - m(还差多少块数)

S2. [产生随机数 V] 生成在 0 到 1 之间均匀分布的一个随机数 V

S3. [生成跳过的条件 p] p = 1 - k/K

S4. [检测] 如果 V < p，则跳转至 S5

S5. [纳入样本] 把下一个记录选为样本，m 和 t 加 1。如果 m < n，则跳转至 S2；否则，算法终止

S6. [跳过，不纳入样本]t++(跳过该块)，K--，p*=1 - k/K，并跳转至 S4

Vitter 算法，是一种蓄水池抽样算法，它的主要实现在 Vitter 发表的 Random sampling with a reservoir 中有介绍，该文首先介绍了 R 算法并分析了 R 算法的时间复杂度，然后在 R 算法的基础上进行改进，主要的改进包括：

1）随机数生成算法改进。

2）通过随机变量随机跳过一些记录，避免了 R 算法的 O(N) 复杂度。

通过改进实现框架（第 3 节），Vitter 介绍了 X 算法和 Y 算法及 Z 算法（第 4 节、第 5 节），在文章的第 6 节对算法进行了再次优化，并在第 8 节给出了算法实现的伪码，PostgreSQL 的 Vitter 算法实现基本和文章中的伪码一致，即取得一个临界点 22.0×n（其中 22.0 是文章中的推荐值，n 为样本容量），在临界点之下通过 X 算法产生随机变量 Z，而在临界点之上，则通过 Z 算法产生随机变量 Z。

采样函数 acquire_sample_rows 的实现框架为第一阶段使用 S 算法进行页面（BLOCK）的选择，在选择到页面之后，借助 Vitter 算法选择需要的元组到“蓄水池”中（reservoir，代码中对应的变量是 rows 数组）。

```
acquire_sample_rows
{
    BlockSampler_Init //S 算法赋初值
    reservoir_init_selection_state //Z 算法赋初值，产生随机变量种子
    while(BlockSampler_HasMore) //当表的块（页面）还没处理完或者蓄水池还没满
    {
        block = BlockSampler_Next;//S 算法的核心实现，随机选择下一个块
```

版权所有，严禁传播，违者自负法律责任！


```
        foreach(tuple in block)
        {
          if(蓄水池未满)
          {
              记录 TUPLE 到蓄水池，先把蓄水池填满
              Continue;
          }
            if(rowstoskip < 0) //rowstoskip 就是 Z 算法中的随机变量 t
                rowstoskip = reservoir_get_next_S;//重新生成随机变量
            if(rowstoskip <= 0)
            随机替换蓄水池中的元组

            rowstoskip --;
        }
    }
}
```

通过两阶段采样算法获得蓄水池中的元组和元组在物理存储上的顺序并不一致，因此需要通过一次排序（qsort_arg + compare_rows）来改变蓄水池中元组的顺序，达到和物理存储一致的效果，如图 5-3 所示。

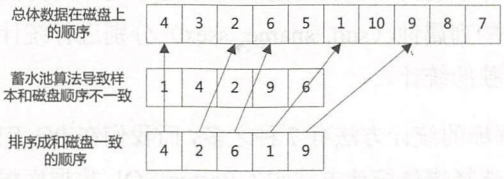


图 5-3　采样数据排成物理顺序

在数据采样的过程中，同时也记录了一些页面和元组的特征，这些特征可以帮助我们进一步估算表中的元组数量（reltuples），如表 5-4 所示。

表 5-4　元组数量估算的相关变量

类型	说明
old_rel_pages	目前采样的表中的总页面数，从PG_CLASS系统表中可以获得
old_rel_tuples	目前采样的表中的总元组数，从PG_CLASS系统表中可以获得
scanned_pages	对页面进行采样（第一阶段）的过程中获得的页面数
scanned_tuples	对元组进行采样（第二阶段）的过程中获得的有效元组数（例如元组上没有DELETE标记）
total_pages	在第一阶段采样时，获得了采样的表中的实际总页面数

版权所有，严禁传播，违者自负法律责任！

我们采用 EMA（指数平均数指标）方法来对表中元组的密度（每个页面有多少元组）进行估计，最终获得最新的元组密度（updated_density）。

```
old_density = old_rel_tuples / old_rel_pages;
new_density = scanned_tuples / scanned_pages;
multiplier = (double) scanned_pages / (double) total_pages;
updated_density = old_density + (new_density - old_density) * multiplier;
```

然后用表中的实际页面数和最新的元组密度相乘，即可获得最新的 reltuples（+0.5 表示 4 舍 5 入）。

```
reltuples = floor(updated_density * total_pages + 0.5);
```

使用 EMA 方法的好处是综合了原有的值（old_rel_tuples）和采样的值两种情况，而不是只采纳采样的值，这样也会带来一些问题。例如将一个数据量比较大的表中的数据全部删除，然后对表做 ANALYZE 操作，这时候通过 EMA 方法获得的 reltuples 就会和实际值的偏差比较大。

5.1.3.2 统计方法

通过两阶段采样获得样本之后，就要对这些样本进行统计，在执行 ANALYZE STUDENT 之后，会对 STUDENT 表的列属性（sno, sname, ssex）分别进行统计，假如 STUDENT 表上有索引，还会对索引进行单独的统计。

目前 PostgreSQL 数据库的统计方法有 7 种之多，而我们在 PG_STATISTIC 表中的槽只有 5 个，那么针对每一列具体选择哪种统计方法呢？PostgreSQL 数据库根据列属性的类型及该类型可以使用的方法来决定采用的统计方法，选择统计方法的代码在 std_typanalyze 函数中实现，依照列属性的类型可以获得该属性对应的操作符表（存储在 PG_OPCLASS 系统表中），依照操作符类结合（PG_AMOP 系统表）可以获得比较的操作符，这种比较操作符和列属性的类型相对应，然后就可以根据是否具有等值操作符和不等值操作符来确定对应列属性的统计方法。

```
// eqopr：等值操作符
// ltopr：小于操作符（代表不等值的情况）
 if (OidIsValid(eqopr) && OidIsValid(ltopr))
  {
      stats->compute_stats = compute_scalar_stats;
      stats->minrows = 300 * attr->attstattarget;
  }
  else if (OidIsValid(eqopr))
  {
```

版权所有，严禁传播，违者自负法律责任！


```
        stats->compute_stats = compute_distinct_stats;
        stats->minrows = 300 * attr->attstattarget;
    }
    else
    {
        stats->compute_stats = compute_trivial_stats;
        stats->minrows = 300 * attr->attstattarget;
    }
```

因为其中 compute_scalar_stats 函数涉及的统计方法比较多，因此下面重点分析 compute_scalar_stats 函数。

以 STUDENT 表中的 sname 属性为例，sname 列属性的值域为{ ‘zs’, ‘ls’, ‘ww’, ‘zl’, ‘zs’, ‘ls’, null}，由于没有达到采样容量的下限值，因此所有数据纳入样本，在 compute_scalar_stats 函数中，通过一系列的数据来描述 sname 列属性的状态，如表 5-5 所示。

表 5-5　统计信息的值

变量	变量值	描述
samplerows	7	样本容量
null_cnt	1	NULL值的个数
nonnull_cnt	6	非NULL值的个数
toowide_cnt	0	超长的值的个数
total_width	18	非NULL值的长度总和，包含变长类型（VARCHAR）的Header长度

compute_scalar_stats 函数还会给 sname 列属性中的值进行编号，如图 5-4 所示，注意图中忽略了 NULL 值。

编号后，对 sname 列属性中的值进行排序，排序后的情况如图 5-5 所示。

Values.tupno 编号	0	1	2	3	4	5
Values.value 值	zs	ls	ww	zl	zs	ls
tupnoLink 去重编号	0	1	2	3	4	5

图 5-4　编号前的情况

Values.tupno 编号	1	5	2	3	0	4
Values.value 值	ls	ls	ww	zl	zs	zs
tupnoLink 去重编号	4	5	2	3	4	5

图 5-5　编号后的情况

在排序过程中对 tupnoLink 中的值也做了处理，主要针对的是其中的重复值，如果有连续的相同值，tupnoLink 记录了最后一个相同值的来源编号，这样就能标记出最后一个相同值在哪。

版权所有，严禁传播，违者自负法律责任！

例如排序后的最后一个'ls'，它排序前在编号 5 的位置，即 tupno[tupnoLink] = tupno[5] == 5，再例如排序后的最后一个'zs'，它排序前在编号 4 的位置，即 tupeno[tupnoLink] = tupno[4] == 4，通过这个标识我们能找到重复值的结尾处，从而方便进行去重（distinct）操作。表 5-6 显示的是在排序之后确定下来的一些统计信息。

表 5-6　统计信息的值

变量	变量值	描述
ndistinct	4	去掉重复值和NULL值之后的个数
stanullfrac	1/7	NULL值比例 = null_cnt / samplerows
stdwidth	3	变长类型 = total_width / nonnull_cnt
stadistinct	4/7	分为3种情况 * 该列无重复值，则stadistinct = -1×(1-stanullfrac) * 该列每个值都有重复值，stadistinct = ndistinct * 其他，stadistinct依照Haas和Stokes的方法估值，利用stadistinct = $n*d / (n - f1 + f1*n/N)$公式进行计算，其中： n代表的是样本容量减去NULL值的数量 d代表的是在其中消除和去除掉NULL值之后的数量 $f1$代表的是在列中完全没有重复的行的数量 N代表的是全体总行数 然后如果stadistinct的值超过了总行数的10%，那么： stadistinct = stadistinct/totalrows

stadistinct 在确定之后是一个整数值，用来代表去重之后有多少个独立的值，但如果整数值的数量比较大，超过了总行数的 10%，那么 stadistinct 中保存的就是小数值。

stadistinct = stadistict /totalrows （超过总行数的 10%）

然后就可以确定要保存多少个 MVC 值，这要分成两种情况，一种情况是所有的值都保存成 MVC 值，另一种是要结合每个值的重复率及直方图的桶容量进行调整。

```
//情况 1：所有的值都可以用来做 MCV 值，这需要满足如下条件
//所有的非 NULL 值都有重复值 --track_cnt == ndistinct
//没有宽列 --toowide_cnt == 0
//去重之后占总行数的比例不能超过 10%  -- stats->stadistinct > 0
//track 没满或刚满
if (track_cnt == ndistinct && toowide_cnt == 0 &&
stats->stadistinct > 0 &&
track_cnt <= num_mcv)
{
```

版权所有，严禁传播，违者自负法律责任！


```
    /* Track list includes all values seen, and all will fit */
    //统计到的桶里的所有的值都记成 MCV
    num_mcv = track_cnt;
}
//情况 2：部分值做 MCV 值
else
{
    double    ndistinct_table = stats->stadistinct;
    double    avgcount,
              mincount,
              maxmincount;

    /* Re-extract estimate of # distinct nonnull values in table */
    if (ndistinct_table < 0)//如果是负小数，换成正数，也就是变成数量而非比例
        ndistinct_table = -ndistinct_table * totalrows;
    /* estimate # occurrences in sample of a typical nonnull value */
    //平均每个非 NULL 值有几个重复值
    avgcount = (double) nonnull_cnt / ndistinct_table;
    /* set minimum threshold count to store a value */
    //要记到 MCV 里的必须比平均值再多 25%
    mincount = avgcount * 1.25;

    //必须多于 1 个，至少 2 个，这里是做矫正
    if (mincount < 2)
        mincount = 2;

    //计算直方图每个桶里平均有几个
    maxmincount = (double) values_cnt / (double) num_bins;

    //这意思就是如果一个值的数量超过了直方图的桶容量，就记成 MCV
    if (mincount > maxmincount)
        mincount = maxmincount;

    //这意思就是看看当前最多有几个 mcv (track_cnt),num_mcv，想要多点也不可能
    if (num_mcv > track_cnt)
        num_mcv = track_cnt;

    //综合最多有几个和 mincount 一起出 MCV 值
    for (i = 0; i < num_mcv; i++)
    {
        if (track[i].count < mincount)
```

版权所有，严禁传播，违者自负法律责任！


```
        {
            num_mcv = i;
            break;
        }
    }
}
```

由于我们的示例的样本空间比较小，只有'zs'和'ls'两个有重复值，因此 MCV 的桶数（num_mcv 变量）为 2，统计的过程比较简单，两个词的词频都是 2/7。

直方图的桶数（num_hist 变量）一方面和系统的默认参数 default_statistics_target 相关，另一方面和 MCV 的桶数直接相关：num_hist = ndistinct – num_mcv，如果 num_hist 的值超出了系统默认设定的范围，则按照系统默认的值统计。PostgreSQL 数据库采用了等频直方图来描述数据的分布情况，它首先去掉在 MCV 中已经存在的值，使用剩余的值进行统计，等频直方图示例如图 5-6 所示。

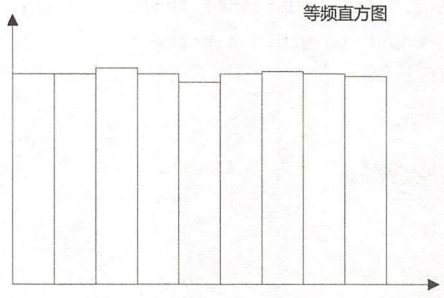


图 5-6　等频直方图（每个桶的高近似相等）

```
//假如有 11 个值：1 1 2 3 4 5 5 6 7 7 8
//需要建 4 个桶，则 delta = 3，deltafrac = 1
//初始情况：  记录 values[0]做边界值，pos = 3，posfrac = 1
//第一轮：    记录 values[pos = 3]做边界值，  pos = 6，posfrac = 2
//第二轮：    记录 values[pos = 6]做边界值，  pos = 9，posfrac = 3，进位:pos = 10，posfrac = 0
//第三轮：    记录 values[pos = 10]做边界值，  pos = 13，posfrac = 1
delta = (nvals - 1) / (num_hist - 1);
deltafrac = (nvals - 1) % (num_hist - 1);
pos = posfrac = 0;

for (i = 0; i < num_hist; i++)
{
    hist_values[i] = datumCopy(values[pos].value,
```

```
    stats->attrtype->typbyval,
    stats->attrtype->typlen);
    pos += delta;
    posfrac += deltafrac;
    if (posfrac >= (num_hist - 1))
    {
        /* fractional part exceeds 1, carry to integer part */
        pos++;
        posfrac -= (num_hist - 1);
    }
}
```

相关系数的产生则依赖于排序的过程，我们对比排序前后的编号变化就可以发现，排序后的顺序发生了错乱，我们使用这两组编号的相关系数来描述这两组数据的相关性，如图 5-7 所示。

Values.tupno 排序前编号	0	1	2	3	4	5
Values.tupno 排序后编号	1	5	2	3	0	4

图 5-7　用来生成相关系数的编号

相关系数的公式为：

$$\rho = \frac{n\sum xy - \sum x \sum y}{\sqrt{n\sum x^2 - (\sum x)^2}\sqrt{n\sum y^2 - (\sum y)^2}}$$

假设排序前的编号为 x，排序后的编号为 y，而编号有如下特点：

$$\sum x = \sum y$$

$$\sqrt{n\sum x^2 - (\sum x)^2} = \sqrt{n\sum y^2 - (\sum y)^2}$$

因此 PostgreSQL 化简了相关系数 ρ 的公式：

$$\rho = \frac{n\sum xy - (\sum x)^2}{n\sum x^2 - (\sum x)^2}$$

相关系数的计算代码如下：

```
∑xy = foreach i (corr_xysum += ((double) i) * ((double) tupno));
∑x = corr_xsum = ((double) (values_cnt - 1)) *((double) values_cnt) / 2.0;
```

```
∑x² = corr_x2sum = ((double) (values_cnt - 1)) *
                   ((double) values_cnt) * (double) (2 * values_cnt - 1) / 6.0;

ρ = corrs[0] = (values_cnt * corr_xysum - corr_xsum * corr_xsum) /
                   (values_cnt * corr_x2sum - corr_xsum * corr_xsum);
```

5.1.4 多列统计信息生成

如果用户通过 CREATE STATISTIC 创建了多列统计信息项，那么在对一个表做单列统计信息时，也会尝试同时调用 BuildRelationExtStatistics 函数创建多列统计信息。

统计模块会根据用户指定的统计类型来生成统计信息，如果用户没有指定具体的统计类型，则统计模块会默认对 STATS_EXT_NDISTINCT 类型和 STATS_EXT_DEPENDENCIES 类型都进行统计。

5.1.4.1 STATS_EXT_NDISTINCT 类型

这种类型的统计信息和单列统计信息中的 stadistinct 是类似的，stadistinct 中记录的是单列中去掉 NULL 值并消重之后的数据量，STATS_EXT_NDISTINCT 类型的统计信息则记录的是基于多列的消重之后的数据量。

例如，对 STUDENT(sno, sname, ssex)创建多列统计信息，那么 STATS_EXT_NDISTINCT 类型的统计信息会生成 4 组数据，分别是{sno, sname}、{sno, ssex}、{sname, ssex}、{sno, sname, ssex}的消重之后的数据量。在 PG_STATISTIC_EXT 系统表中的 stxndistinct 中保存的数据为{"1, 2": 7, "1, 3": 7, "2, 3": 7, "1, 2, 3": 7}，其中的{"2, 3": 7}代表的就是对{sname, ssex}消重之后数据量是 7。

```
--对{sname, ssex}做 GROUP BY 操作
postgres=# SELECT COUNT(*) FROM (SELECT sname, ssex FROM STUDENT GROUP BY sname, ssex) st;
 count
-------
     7
(1 row)
```

STATS_EXT_NDISTINCT 类型的多列统计信息主要在 statext_ndistinct_build 函数中生成，它的主要流程如下：

```
//numattrs 是多列统计信息要统计的总列数
//k 是每轮要选择 k 个列来生成统计信息，最少为两列生成统计信息
//因此 k 从 2 开始
for (k = 2; k <= numattrs; k++)
```

```
{
    int         *combination;
    CombinationGenerator *generator;

    //从numattrs中选择k个列，共生成C(numattrs, k)种情况
    generator = generator_init(numattrs, k);

    //为每一种情况生成统计信息
    while ((combination = generator_next(generator)))
    {
        ......
        //获得要生成统计信息的列的单列统计信息
        //例如要对{sname, ssex}生成统计信息，这里需要获得sname的单列统计信息
        //和ssex的单列统计信息，注意，这些单列统计信息是在ANALYZE操作中传递过来的
        for (j = 0; j < k; j++)
            item->attrs = bms_add_member(item->attrs,
                                         stats[combination[j]]->attr->attnum);

        //排序，并终借用n*d / (n - f1 + f1*n/N)公式进行估算
        item->ndistinct =
            ndistinct_for_combination(totalrows, numrows, rows,
                                      stats, k, combination);

        itemcnt++;
    }

    generator_free(generator);
}
```

5.1.4.2　STATS_EXT_DEPENDENCIES 类型

在关系数据库中，函数依赖（functional dependency）是对一个表（关系）上的两个列（属性）进行描述的一种方法：如果对于 X 属性中的每一个值在 Y 属性上有且只有一个和它一一对应的值，那么就说 Y 对 X 满足函数依赖关系，写做 X -> Y。例如对于 STUDENT 表，因为 sno 上有主键索引，因此每一个 sno 都和一个 sname 一一对应，因此 sname 对 sno 满足函数依赖关系，也就是 sno - > sname。需要注意的是，sname 对 sno 满足函数依赖关系，反过来不一定成立，也就是说 sno 对 sname 不一定满足函数依赖关系。

在实际的应用中，有很多属性之间并不严格的满足函数依赖关系，对于多列统计信息而言，PostgreSQL 数据库建立了一种“软”函数依赖关系，它会记录一个“函数依赖度”。

函数依赖度的定义是：两个属性之间满足函数依赖的值占总体数量的比例，当然这也可以扩展到多个属性。比如对于"1, 3 => 2": 1.000000 代表的就是{sno, ssex} -> {sname}中满足函数依赖的比例是 1，也就是全部满足。函数依赖度的获取是通过 dependency_degree 函数实现的。

```
//例如要计算{sname, ssex} -> {sno}的函数依赖度
//这时候所有的值已经排序完毕，排序完成之后的数据是按照 sname、ssex、sno 排列的
for (i = 1; i <= numrows; i++)
{
    //注意 multi_sort_compare_dims 的参数
    //0 和 k-2 表示了前 k-1 个值，对我们的例子而言就是{sname, ssex}
    //items[i - 1]和 items[i]，代表当前值和上一个值
    //这里要判断的是{sname,ssex}的当前值和上一个值是否相同，
    //如果判断结果不相同，则有可能发现了一个满足函数依赖的值
    if (i == numrows ||
        multi_sort_compare_dims(0, k - 2, &items[i - 1], &items[i], mss) != 0)
    {
        //如果当前值和上一个值不同，而且没有违背过函数依赖关系，
        //那么把满足函数依赖关系的值计入总数
        if (n_violations == 0)
            n_supporting_rows += group_size;

        //违背依赖关系的标记清空，满足依赖关系的计数器清空
        n_violations = 0;
        group_size = 1;
        continue;
    }
    //如果判断结果相同，那么再判断一下两个属性的第 k 个值是否相同
    //对我们的示例而言就是要再判断一下 ssex 是否相同
    //如果判断结果相同，那么我们认为它仍然满足函数依赖关系
    //如果判断结果不相同，那么就违背了函数依赖关系，通过 n_violations 把它记下来
    else if (multi_sort_compare_dim(k - 1, &items[i - 1], &items[i], mss) != 0)
        n_violations++;

    //如果前 k-1 个值都相同，第 k 个值也相同，那么它也满足函数依赖关系，这里进行计数
    group_size++;
}
```

假设 STUDENT 表中的{sname, sno}的数据已经排序完成，如果要计算 sname -> sno 的函数依赖度，计算过程如图 5-8 所示，在步骤 7 获得了一共 3 个满足函数依赖的项，则可以计算函数依赖度为 3/7 = 0.428571。

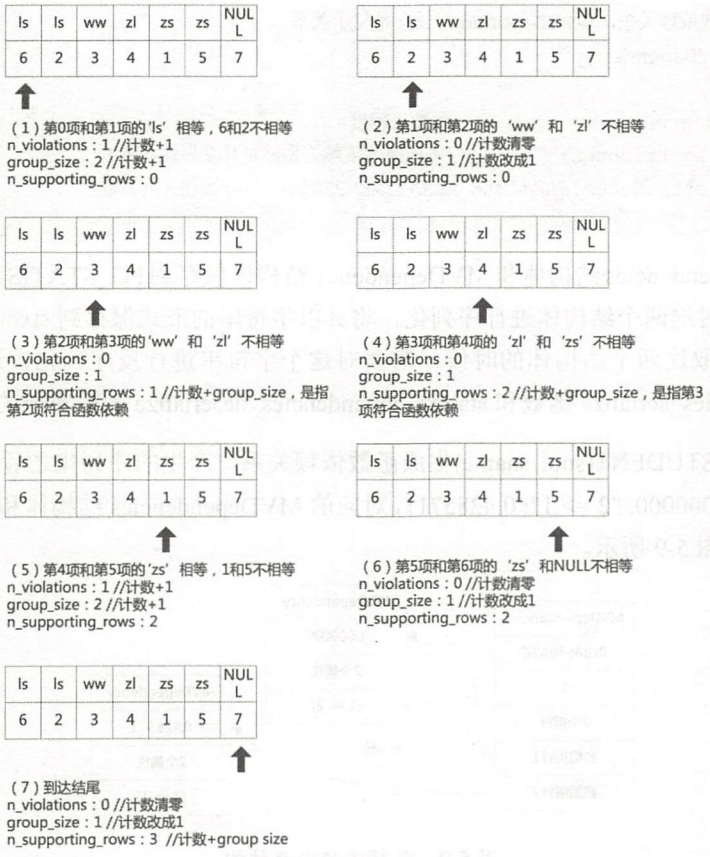


图 5-8　函数依赖度的生成流程

函数依赖关系在生成的过程中使用 MVDependencies 结构体和 MVDependency 结构体保存依赖关系，我们先来认识这两个结构体。

```
//记录整个多列统计信息生成的函数依赖关系
typedef struct MVDependencies
{
   uint32      magic;          //目前是 STATS_DEPS_MAGIC
   uint32      type;           //目前是 STATS_DEPS_TYPE_BASIC
   uint32      ndeps;          //产生了几个 MVDependency

   //MVDependency 数组，分别记录每一个单独的函数依赖关系
   MVDependency *deps[FLEXIBLE_ARRAY_MEMBER];    /* dependencies */
} MVDependencies;
```

```
//记录每个单独的函数依赖关系，和MVDependencies是父子关系
typedef struct MVDependency
{
   double      degree;              //函数依赖度
   AttrNumber   nattributes;        //本项函数依赖关系涉及几个属性（列）
   AttrNumber   attributes[FLEXIBLE_ARRAY_MEMBER];    //属性（列）数组
} MVDependency;
```

当把 MVDependencies 结构体和 MVDependency 结构体保存到 PG_STATISTIC_EXT 系统表中的时候，需要对这两个结构体进行序列化，将其以字符串的形式保存到 stxdependencies 列。当从系统表中读取这两个结构体的时候，需要对这个字符串进行反序列化，这个过程是通过 statext_dependencies_serialize 函数和 statext_dependencies_deserialize 函数实现的。

例如，对于 STUDENT(sno, sname)生成函数依赖关系，产生的序列化之后的字符串是这样的：{"1 => 2": 1.000000, "2 => 1": 0.428571}，对应的 MVDependencies 结构体和 MVDependency 结构体的形式如图 5-9 所示。

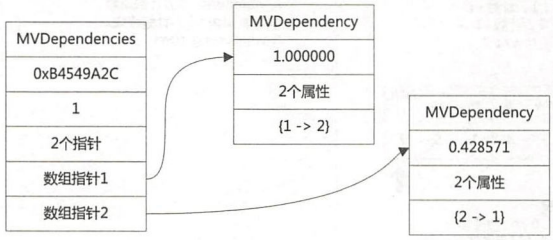


图 5-9　函数依赖内存结构

5.2　选择率

前面已经介绍过选择率的含义，它是通过约束条件过滤之后保留的元组数占约束条件过滤之前的元组数的比例，选择率的估计需要借助于统计信息，对统计信息我们已经进行了介绍，它包括直方图、高频值、NULL 值率等，我们可以根据这些特征来估算选择率。

例如，对 STUDENT 表生成的统计信息就能在对 STUDENT 表进行查询的时候作为计算选择率的依据。假如执行 SQL 语句 SELECT * FROM STUDENT WHERE sname = 'ww' AND (ssex IS NOT NULL OR sno > 5)，其中 sname = 'ww' AND (ssex IS NOT NULL OR sno > 5)是由 3 个子约束条件拼接起来的一个完整的约束条件，对于这个约束条件，会分别计算（sname = 'ww'）、（ssex IS NOT NULL）、（sno > 5）三个子约束条件的选择率，然后根据其中的 AND 运算符

和 OR 运算符再计算总的选择率。

其中（sname = 'ww'）的选择率的计算过程如下：

- 获得 sname 列对应的 stanullfrac = 0.142857。
- 获得高频值数组{0.285714,0.285714}，计算高频值的总比例 = 0.285714＋0.285714 = 0.571428。
- 因为 sname='ww'既不是高频值，也不是 NULL 值，所有的元组的总比例是 1，因此可以先去除 NULL 值和高频值，计算其余的元组所占的比例 = 1 - 0.142857 - 0.571428 = 0.285714。
- 计算除了 NULL 值和高频值，剩下还有几个值：其中元组数量 = 7，stadistinct = -0.571429，可以获得去除 NULL 值并消重之后，还剩 7×0.571429 = 4 个元组，这 4 个元组中还有两个高频值，从 4 个元组中去掉两个高频值，也就是说还有两个值。
- 假设两个值平均分配选择率，可以获得 sname = 'ww'的选择率是 0.285714/2 = 0.142857。

其中（ssex IS NOT NULL）的选择率的计算过程如下：

- 获得 ssex 列对应的 stanullfrac = 0.142857，这些对应的是 NULL 值的选择率。
- 非 NULL 值的选择率即为 1 - stanullfrac = 1 - 0.142857 = 0.857142。

其中（sno > 5）的选择率计算过程如下：

- 根据 sno 属性的直方图{1,2,3,4,5,6,7}计算 sno <= 5 的选择率，因为直方图是左闭右开区间，即[1，2)，[2，3)...因此 sno <5 共占了 4 个桶，共有 6 个桶，所以选择率是 0.6666666666666666，虽然 sno=5 则位于[5，6)这个桶内（也就是说位于第 5 个桶），但由于 5 是边界值，所以这部分选择率没有被计算入内。
- 根据 sno <= 5 的选择率计算 sno > 5 的选择率 = 1 – 0.6666666666666666 = 0.3333333333333333。

在计算了每个子约束条件独立的选择率之后，就可以根据 AND 运算符和 OR 运算符计算它们的综合的选择率，AND 运算符和 OR 运算符的选择率计算是基于概率的，已知基于独立事件的概率的加法和乘法的公式为：

$$P(A+B) = P(A) + P(B) - P(AB)$$

$$P(AB) = P(A) \times P(B)$$

可以首先获得约束条件(ssex IS NOT NULL OR sno > 5)的选择率为：

$$
\begin{aligned}
&P(\text{ssex IS NOT NULL OR sno} > 5)\\
&= P(\text{ssex IS NOT NULL}) + P(\text{sno} > 5) - P(\text{ssex IS NOT NULL AND sno} > 5)\\
&= P(\text{ssex IS NOT NULL}) + P(\text{sno} > 5) - P(\text{ssex IS NOT NULL}) \times P(\text{no} > 5)\\
&= 0.857142 + 0.333333 - 0.857142 \times 0.333333\\
&= 0.90476
\end{aligned}
$$

然后可以获得 sname = 'ww' AND (ssex IS NOT NULL OR sno > 5)的总的选择率为：

$$
\begin{aligned}
&P(\text{sname = 'ww' AND (ssex IS NOT NULL OR sno} > 5))\\
&= P(\text{sname = 'ww'}) \times P(\text{ssex IS NOT NULL OR sno} > 5)\\
&= 0.142857 \times 0.90476\\
&= 0.129252
\end{aligned}
$$

因为STUDENT表中目前有7个元组，因此这个查询最终可能获得的结果集有 $7 \times 0.129252 \approx 1$ 条元组。

示例中的计算过程比较简单，是因为在计算的过程中忽略了一些情况。例如对于既有高频值和直方图的统计信息的列，就需要同时考虑高频值和直方图，例子中的 sno > 5 没有高频值信息，只有直方图信息，所以在计算的时候就直接使用直方图就可以了，但是在实际情况中，计算的过程会比示例中复杂一些。

另外，根据示例中计算出来的选择率可以估计出查询结果是一条元组，而实际情况确实只有一条元组，这样的结果虽然看上去令人欣喜，但是请保持冷静，这里示例只是一个特例，对这样的特例的结果并不能推演出一个普遍的结论，因此不能草率地说目前选择率的计算结果一定是准确的。需要谨记的是，选择率仍然是一个估计值，因为首先我们不能保证统计信息的准确性，统计信息是基于样本的（我们的示例过于简单，样本是整个表的数据，对于数据量比较大的表，样本只是表中的一部分数据），样本是否显著对统计信息的结果有很大的影响。对一个表进行更新之后，不会立即把所有的统计信息都同步更新一遍，这时选择率的计算还依赖于旧的统计信息。另外，基于概率的计算方法也不适用于所有的情况，例如对于这样一个 SQL 语句：

```
SELECT * FROM STUDENT WHERE sno = 7 AND sname IS NULL AND ssex IS NULL;
```

通过统计信息可以快速地给出（sno = 7 AND sname IS NULL AND ssex IS NULL）的选择率：

$$
\begin{aligned}
P(\text{sno} = 7) &= 0.142857\\
P(\text{sname IS NULL}) &= 0.142857\\
P(\text{ssex IS NULL}) &= 0.142857
\end{aligned}
$$

$$
\begin{aligned}
&P(\text{sno} = 7 \text{ AND sname IS NULL AND ssex IS NULL}) \\
&\quad = P(\text{sno} = 7) \times P(\text{sname IS NULL}) \times P(\text{ssex IS NULL}) \\
&\quad = 0.142857 \times 0.142857 \times 0.142857 \\
&\quad = 0.002915
\end{aligned}
$$

计算获得的选择率为 0.002915，约为千分之三，这样的选择率显然不能代表语句的真实选择率，这时候如果 STUDENT 表上有多列统计信息，就有可能应用到这个选择率的计算上，可能结果会更准确一些。

另外，统计信息也并不能覆盖选择率计算的所有情况，并不是所有的约束条件都能使用统计信息进行选择率计算。例如带有 Param 的约束条件，如果 Param 不是常量值，就没有计算选择率的依据，这时候只能设置一个默认值，PostgreSQL 数据库设定了大量的默认值，我们把它列到表 5-7 中。

表 5-7　部分选择率的默认值

变量名	值	说明
DEFAULT_EQ_SEL	0.005	等值约束条件的默认选择率，例如 A = b
DEFAULT_INEQ_SEL	0.3333333333333333	不等值约束条件的默认选择率，例如 A < b
DEFAULT_RANGE_INEQ_SEL	0.005	涉及同一个属性（列）的范围约束条件的默认选择率，例如A > b AND A < c
DEFAULT_MATCH_SEL	0.005	基于模式匹配的约束条件的默认选择率，例如LIKE
DEFAULT_NUM_DISTINCT	200	对一个属性消重（distinct）之后的值域中有多少个元素，通常和DEFAULT_EQ_SEL互为倒数
DEFAULT_UNK_SEL	0.005	对BoolTest或NullText这种约束条件的默认选择率，例如IS TRUE或IS NULL
DEFAULT_NOT_UNK_SEL	(1.0 - DEFAULT_UNK_SEL)	对BoolTest或NullText这种约束条件的默认选择率，例如IS NOT TRUE或IS NOT NULL

PostgreSQL 数据库选择率的计算函数为 clauselist_selectivity 函数，它的参数是基于合取范式的约束条件链表，在 clauselist_selectivity 函数中，会遍历约束条件链表中的每个子约束条件，

计算每个子约束条件的选择率，并且通过设定这些子约束条件是独立事件，使用概率的方法获得总的选择率。

clauselist_selectivity 函数中主要考虑了 3 种情况。

- 如果一个合区范式的约束条件都引用了同一个表，而且全部都是 Var = Var 或 Var = Const 这种形式的，那么就可以考虑使用多列统计信息进行选择率的计算。
- 如果无法使用多列统计信息进行选择率的计算，那么对每个子约束条件计算选择率，然后使用概率的方法对所有子约束条件的选择率进行“汇总”。
- 对于 Var > Const 或 Var < Const 这种类型的选择率，不能简单地依赖概率的方法，还需要进行特殊的处理，例如对于 sno > 5 and sno > 6 这样的约束条件，它的子约束条件有包含关系，一个约束条件是另一个约束条件的真子集，这时只需要按照 sno > 6 的选择率计算。

5.2.1 使用函数依赖计算选择率

这里主要是应用函数依赖类型的统计信息，在创建 RelOptInfo 结构体的时候，PostgreSQL 数据库会把多列统计信息加载到 RelOptInfo->statlist 中，这样在对约束条件计算选择率的时候，就可以直接使用这些统计信息。

函数依赖类型的统计信息要想应用于选择率的计算，其对约束条件的形式的限制还是比较多的，最基本的限制是约束条件中只能涉及一个表，而且这个表上有基于函数依赖的统计信息，另外约束条件中的每个子约束条件，也通过 dependency_is_compatible_clause 函数进行了检查：

- 只能是 RestrictInfo 结构体形式的约束条件。
- 不能是常量约束条件（RestrictInfo->->pseudoconstant）。
- 约束条件中只能引用一个表。
- 必须是操作符表达式（OPExpr），且必须是 Var = Const 形式或者 RelabelType = Const 形式。
- Var 必须是用户定义的列，不能是系统伪列。

如果一个表有多个函数依赖类型的统计信息，还需要找到一个最适合当前情况的统计信息，这个比较的过程基于如下原则：

- 约束条件的属性和函数依赖关系统计信息中的键值取交集，交集中的键值多的最适用。
- 如果交集中键值数相同，则两个函数依赖关系统计信息中键值数少的获胜。

例如，对于表 TEST_A(A, B, C, D, E)，如果有 3 个函数依赖类型的统计信息，分别建立在

{A, B}，{A, B, C}，{B, C, D}属性（键）上，则对于约束条件 A = 1 AND B = 1 最适合使用的是{A, B}属性上的统计信息，如表 5-8 所示。

表 5-8　统计信息的使用选择

统计信息的键值	约束条件A = 1 AND B = 1
{A, B}	交集 = {A, B}，统计信息的键值有2个
{A, B, C}	交集 = {A, B}，统计信息的键值有3个
{B, C, D}	交集 = {B}，统计信息的键值有3个

下面分析一下 choose_best_statistics 的流程。

```
foreach(lc, stats)
{
    ......
    //这里需要的是函数依赖类型的统计信息
    if (info->kind != requiredkind)
        continue;

    //attnums：约束条件中引用的属性集合
    //info->keys：统计信息中的键值（也就是值统计信息是建立在哪些属性上的）
    matched = bms_intersect(attnums, info->keys); //取交集
    num_matched = bms_num_members(matched);//交集中的属性数量
    bms_free(matched);

    numkeys = bms_num_members(info->keys);

    //约束条件的属性和函数依赖关系统计信息中的键值取交集，交集中的键值多的最适用
    if (num_matched > best_num_matched ||
        //如果交集中键值数相同，则两个函数依赖关系统计信息中键值数少的获胜
        (num_matched == best_num_matched && numkeys < best_match_keys))
    {
        //记录新的优胜值
        best_match = info;
        best_num_matched = num_matched;
        best_match_keys = numkeys;
    }
}
```

在获得了适用的统计信息之后，就开始使用统计信息计算选择率，假设有约束条件 A = Const AND B = Const，则基于函数依赖类型统计信息的选择率计算公式如下：

$$P(A = Const, B = Const) = P(A = Const) * (d + (1-d) * P(B = Const))$$

（其中 d 是{A -> B}的函数依赖度）

对于 A = Const AND B = Const AND C = Const 这样的约束条件，可以递归求解，先采用{A, B -> C}的函数依赖关系进行计算。

$$P(A = Const\ B = Const, C = Const) = P(A = Const, B= Const) * (d + (1-d) * P(C = Const))$$

（其中 d 是{A, B -> C}的函数依赖度）

然后再计算 *P*(A = Const, B = Const)，通过递归计算，我们可以求解含有任意多个子约束条件的约束条件的选择率（当然需要每一个递归层次上都有对应的函数依赖关系的统计信息）。

另外，对于 A = Const AND B = Const 这样的约束条件，我们在求解的时候既可以使用{A->B}这种函数依赖关系，也可以使用{B->A}这种函数依赖关系，PostgreSQL 数据库设定了如下规则：

- 约束条件中的所有属性应该能覆盖统计信息中的键值（属性）。
- 在覆盖的情况下，选择键值最多的那组统计信息。
- 如果两个统计信息键值数相同，则选择函数依赖度高的统计信息。

find_strongest_dependency 负责找到合适的函数依赖类型的统计信息，它的实现流程如下。

```
//每一条函数依赖类型的统计信息中，都会生成很多项
//MVDependencies 代表一条统计信息，MVDependency 则是统计信息中的每一个函数依赖项
//这部分代码的作用就是从函数依赖项中选出最优的
for (i = 0; i < dependencies->ndeps; i++)
{
   //获得一条函数依赖统计信息项
   MVDependency *dependency = dependencies->deps[i];

   //如果函数依赖统计信息中的键值比约束条件中的键值多，则不能使用
   if (dependency->nattributes > nattnums)
      continue;

   if (strongest)  //strongest 已经存在，那就需要进行比较了
   {
      //用当前的统计信息的键值和最优的统计信息的键值相比较，键值多的获胜
      if (dependency->nattributes < strongest->nattributes)
         continue;
```


非卖品，仅供非商业用途或交流学习使用

```
        //如果键值数一样，则比较函数依赖度，函数依赖度高的获胜
        if (strongest->nattributes == dependency->nattributes &&
            strongest->degree > dependency->degree)
            continue;
    }

    //dependency_is_fully_matched 检查函数依赖项中的键值
    //是否是约束条件中属性集合的子集
    if (dependency_is_fully_matched(dependency, attnums))
        strongest = dependency; /* save new best match */
}
```

例如有 sno = 1 AND sname ='ls' AND ssex = 1 这样的约束条件和建立在{sno, sname}上的函数依赖统计信息{"1 => 2": 1.000000, "2 => 1": 0.428571}，find_strongest_dependency 函数会分别对{"1 => 2": 1.000000 和{"2 => 1": 0.428571}进行匹配。通过分析会发现{sno, sname}是约束条件中属性集合{sno, sname, ssex}的子集，因此{"1 => 2": 1.000000}和{"2 => 1": 0.428571}都符合要求，但是二者的函数依赖度不同，函数依赖度高的获胜，因此最终会选择{"1 => 2": 1.000000}。在获得属性之间的函数依赖度之后就可以利用这个函数依赖关系进行相关约束条件选择率的计算了。

```
//利用函数依赖度计算选择率
//假设约束条件是 A = 1 AND B = 1 AND C = 1 AND D = 1
//函数依赖是建立在{A, B, C}上的
//while 循环的第一轮是计算 P(A = ?, B = ?, C = ?)
//while 循环的第二轮是计算 P(A = ?, B = ?)
//注意 D = 1 这个子约束条件不会在这里求选择率，因为统计信息中没有该属性
//注意 A = 1 这个子约束条件不会在这里求选择率，因为无法对单个的属性应用函数依赖度
//estimatedclauses 变量中不会记录 A = 1 和 D = 1
while (true)
{
    ......

    //函数依赖关系中包含{AB->C}、{AC->B}、{BC->A}
    //注意类似{A->B}也是满足的，但{A->B}不如{AB->C}好，也就是说用当前的
    //统计信息的键值和最优的统计信息的键值相比较，键值多的获胜
    //选择函数依赖度高的那个作为计算选择率的依据，假设选择的是{AB->C}
    dependency = find_strongest_dependency(stat, dependencies,
                                           clauses_attnums);
    ......
```


非卖品，仅供非商业用途或交流学习使用

```
    //开始计算选择率，循环遍历每个子约束条件
    //目的是找到{AB->C}中的C对应的子约束条件
    listidx = -1;
    foreach(l, clauses)
    {
        ......
        //找到{AB->C}中的C对应的子约束条件
        if (dependency_implies_attribute(dependency,
                                  list_attnums[listidx]))
        {
            clause = (Node *) lfirst(l);
            //计算C对应的子约束条件的选择率
            s2 = clause_selectivity(root, clause, varRelid, jointype,
                                    sjinfo);

            //已经计算过的子约束条件记录下来
            *estimatedclauses = bms_add_member(*estimatedclauses, listidx);

            //在约束条件属性集合中把C去掉，
            //去掉之后再次进行while循环，就可以计算P(A = ?, B = ?)了
            clauses_attnums = bms_del_member(clauses_attnums,
                                        list_attnums[listidx]);
        }
    }

    // 函数依赖类型的选择率计算公式：P(A = ?, B = ?) = P(A = ?) * (d + (1-d) * P(B = ?))
    s1 *= (dependency->degree + (1 - dependency->degree) * s2);
}
```

5.2.2 子约束条件的选择率

子约束条件的选择率在 clause_selectivity 函数中计算，clause_selectivity 函数首先给出了默认选择率是 0.5，然后根据子约束条件的不同情况，对子约束条件进行分别处理，对于无法处理的情况，则使用默认选择率 0.5。

在生成物理路径的过程中，由于要尝试生成多种类型的路径，会频繁地对同一个约束条件的选择率求值，因此在 RestrictInfo 结构体中用 RestrictInfo->norm_selec 和 RestrictInfo->outer_selec 来分别保存内连接情况下和外连接情况下的选择率，这两个值符合如下的规则。

- < 0：没有对已经计算过的选择率进行缓存。

非卖品，仅供非商业用途或交流学习使用

- > 1：这是一个冗余的约束条件。
- 其他：缓存了之前计算过的选择率，可以重复使用这个值，避免重复计算。

由于目前物理路径的种类中增加了参数化路径（parameterized path），因此有些约束条件计算的选择率不能缓存，查询优化器通过 clauselist_selectivity 函数的 varRelid 来区分这种能够缓存的情况。

```
//varRelid == 0 或者
//约束条件中只引用了 varRelid
//代表可以重复使用之前缓存的选择率
if (varRelid == 0 || bms_is_subset_singleton(rinfo->clause_relids, varRelid))
{
    if (jointype == JOIN_INNER)
    {
        if (rinfo->norm_selec >= 0) //>= 0 代表记录了之前计算的选择率
            return rinfo->norm_selec;
    }
    else
    {
        if (rinfo->outer_selec >= 0) //>= 0 代表记录了之前计算的选择率
            return rinfo->outer_selec;
    }

    //如果是第一次计算选择率，那么标记这个选择率是可以缓存的
    cacheable = true;
}
```

不同类型的子约束条件，PostgreSQL 数据库的处理方式也不一样，具体的情况如表 5-9 所示。

表 5-9　约束条件对应的选择率计算方法

子约束条件类型	说明
Var	Var通常代表的是表中的一个属性（列），如果它单独作为一个约束条件，通常是如下情况： CREATE TABLE TEST_BOOL(a BOOLEAN, b TEXT); SELECT * FROM TEST_BOOL WHERE a; 这时表TEST_BOOL中的属性a就作为一个单独的约束条件出现在WHERE子句中，PostgreSQL数据库通过boolvarsel函数来获得这一类约束条件的选择率，因为可以将约束条件a改写为a = true，如果a属性有统计信息，则根据统计信息计算a = true的选择率，如果没有统计信息，那么就直接设定默认的选择率0.5

非卖品，仅供非商业用途或交流学习使用

续表

子约束条件类型	说明
Const	Const作为约束条件，选择率的获取需要满足如下规则： • 如果Const常量是NULL值，则代表选择率为0，即所有的元组都不会被选择 • 如果Const常量的值是0，则代表选择率为0，即所有的元组都不会被选择 • 其他情况下，选择率为1，也就是所有的元组都会被选择
Param	对于Param变量，先使用estimate_expression_value进行常量替换，如果可以改变成常量（Const），则按照Const作为约束条件的方式获取选择率，否则默认选择率为0.5
NOT运算符	$P(B) = 1 - P(A)$ {B = NOT A}
AND运算符	$P(AB) = P(A) \times P(B)$ {A AND B}
OR运算符	$P(A + B) = P(A) + P(B) - P(AB)$ {A OR B}
OpExpr	如果是连接条件，调用join_selectivity函数处理 如果是过滤条件，调用restriction_selectivity处理
ScalarArrayOpExpr	调用ScalarArrayOpExpr函数处理
RowCompareExpr	可以转化为OpExpr的选择率求值方式
NullTest	1. 统计信息中包含stanullfrac（NULL值比例，用freq_null变量表示） IS NULL的选择率：freq_null IS NOT NULL的选择率：1.0 - freq_null 2. 没有统计信息 IS NULL的选择率：DEFAULT_UNK_SEL IS NOT NULL的选择率：DEFAULT_NOT_UNK_SEL
BooleanTest	1. 统计信息中包含stanullfrac（NULL值比例，用freq_null变量表示）和高频值数组，高频值数组中最多有两个元组，即true和false，如果数组中的第一个是true，那么true所占的比例就是数组中的值（freq_true），如果高频值数组中的第一个是false，那么freq_true所占的比例就是freq_true = 1.0 – freq_false – freq_null。 IS UNKOWN的选择率：freq_null IS NOT UNKOWN的选择率：1.0 - freq_null IS TRUE的选择率：freq_true IS NOT TRUE的选择率： 1.0 - freq_true IS FALSE的选择率：freq_false IS NOT FALSE的选择率：1.0 - freq_false 2. 统计信息中包含stanullfrac（NULL值比例NULL值比例，用freq_null变量表示），但没有高频值数组。 IS UNKOWN的选择率：freq_null IS NOT UNKOWN的选择率：1.0 - freq_null

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

续表

子约束条件类型	说明
	IS TRUE的选择率：(1.0 - freq_null) / 2.0 IS NOT TRUE的选择率：(freq_null + 1.0) / 2.0 IS FALSE的选择率：(1.0 - freq_null) / 2.0 IS NOT FALSE的选择率：(freq_null + 1.0) / 2.0 3. 没有统计信息 IS UNKOWN的选择率：DEFAULT_UNK_SEL IS NOT UNKOWN的选择率：DEFAULT_NOT_UNK_SEL IS TRUE的选择率：clause_selectivity（BooleanTest->arg） IS NOT TRUE的选择率：1.0 - clause_selectivity（BooleanTest->arg） IS FALSE的选择率：1.0 - clause_selectivity（BooleanTest->arg） IS NOT FALSE的选择率：clause_selectivity（BooleanTest->arg）
CurrentOfExpr	CurrentOfExpr表达式最多返回1个元组，因此选择率是（1/表的元组数量）
RelabelType	递归调用clause_selectivity函数，处理RelabelType ->arg
CoerceToDomain	递归调用clause_selectivity函数，处理CoerceToDomain->arg
其他	统一调用boolvarsel处理

5.2.3　基于范围的约束条件的选择率修正

一个 SQL 语句中，可能出现形如 A > 5 AND A > 7 AND A < 9 AND A < 8 的约束条件，如果按照概率的方法计算选择率就是 $P(A > 5) \times P(A > 7) \times P(A < 9) \times P(A < 8)$，但是这个约束条件的子约束条件有重叠的部分，$P(A > 5)$、$P(A > 7)$、$P(A < 9)$、$P(A < 8)$并不是独立的约束条件，因此这样计算的结果肯定是错误的。

PostgreSQL 数据库如果发现子约束条件是操作符表达式而且操作符是‘<’或者‘>’，那么它就不急于计算 $P(A > 5) \times P(A > 7) \times P(A < 9) \times P(A < 8)$，而是先将这个子约束条件的选择率保存到 RangeQueryClause 类型的链表中。

```
//RangeQueryClause 是一个链表，链表的每个节点代表一个属性（列，Var）
typedef struct RangeQueryClause
{
   struct RangeQueryClause *next;          //链表指针
   Node      *var;                         //代表一个属性（列，Var）
   bool       have_lobound;                //约束条件中和 var 相关的条件的有下界
   bool       have_hibound;                //约束条件中和 var 相关的条件的有上界
   Selectivity lobound;                    //var 的下界对应的子约束条件的选择率
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
    Selectivity hibound;                    //var 的上界对应的子约束条件的选择率
} RangeQueryClause;
```

例如约束条件 A > 5 AND A > 7 AND A < 9 AND A < 8 中只引用了一个属性 A，因此它的 RangeQueryClause 类型的链表中只有一个节点，从 $P(A < 9) \times P(A < 8)$中可以看出这个约束条件的上界是 8，从 $P(A > 5) \times P(A > 7)$中可以看出约束条件的下界是 7，因此可以获得上界的选择率是 $P(A < 8)$，下界的选择率是 $P(A > 7)$。

再例如约束条件 A > 5 AND A > 7 AND A < 8 AND B > 1 AND B > 4 AND B >10，它里面引用了两个属性，分别是 A 和 B，RangeQueryClause 类型的链表中最终会有两个节点，分别记录属性 A 和属性 B 的上界和下界的信息。对于 A 属性可以知道它的上界是 8，下界是 7，因此它需要记录 $P(A > 7)$的下界选择率和 $P(A < 8)$的上界选择率，对于 B 属性可以知道它的下界是 10，但是没有上界，因此只记录它的下界选择率 $P(B > 10)$。这种基于范围的约束条件的选择率修正是在 clauselist_selectivity 函数的结尾处进行的。

```
//循环处理每一个 RangeQueryClause 节点
while (rqlist != NULL)
{
    RangeQueryClause *rqnext;

    //如果上界和下界同时存在
    if (rqlist->have_lobound && rqlist->have_hibound)
    {
        ......
        //如果上界和下界都是默认值，做一下特殊处理
        if (rqlist->hibound == DEFAULT_INEQ_SEL ||
            rqlist->lobound == DEFAULT_INEQ_SEL)
        {
        //基于范围的约束条件的选择率默认值
            s2 = DEFAULT_RANGE_INEQ_SEL;
        }
        else
        {
        //计算选择率，P(A > x) + P(A < y) = 1 - P(A > x AND A < y)
            s2 = rqlist->hibound + rqlist->lobound - 1.0;

        //考虑 NULL 值
        //P(A > x) = 1 - P(A <= x) - p(NULL)
        //P(A < y) = 1 - P(A >= y) - p(NULL)
        //   P(A > x) + P(A < y)
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
            //= 1.0 - P(A <= x) - p(NULL) + 1.0 - P(A >= y) - p(NULL)
            //注意算式中的两个 P(NULL),说明 P(NULL)被重复减去了一次
                s2 += nulltestsel(root, IS_NULL, rqlist->var,
                                  varRelid, jointype, sjinfo);
                ......
            }
            //将选择率整合进总选择率
            s1 *= s2;
        }
        else
        {
            //如果只有上界或者下界，则直接使用上界或者下界的的选择率整合进总选择率
            if (rqlist->have_lobound)
                s1 *= rqlist->lobound;
            else
                s1 *= rqlist->hibound;
        }
        //进入下一个属性
        rqnext = rqlist->next;
        pfree(rqlist);
        rqlist = rqnext;
    }
```

5.3　OpExpr 的选择率

OpExpr 这种类型的约束条件是比较常用的约束条件，treat_as_join_clause 函数对这种约束条件进行了分类，分成了连接条件和过滤条件（注意这里的连接条件和过滤条件和我们在第 4 章中介绍的连接条件和过滤条件的区分方法略有不同，这里主要是基于选择率计算的需要）。

- 如果 varRelid 有值，有可能是一个参数化路径，这时认为 OpExpr 是过滤条件。
- 如果 sjinfo == NULL，可能是生成扫描路径，这时认为 OpExpr 是过滤条件。
- 其他情况：如果 OpExpr 中只引用了一个表，按照过滤条件的方法来获得选择率，如果 OpExpr 中引用了多个表，则按照连接条件来获取选择率。

如果是过滤条件就调用 restriction_selectivity 函数来获得 OpExpr 表达式的选择率，如果是连接条件则调用 join_selectivity 函数来获得选择率。下面分别来看一下 restriction_selectivity 函数和 join_selectivity 是如何获得选择率的。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

在 PostgreSQL 数据库中，操作符保存在 PG_OPERATOR 系统表中，PG_OPERATER 系统表有两个属性（列）。

- oprrest：这个属性指定一个函数，把 OpExpr 当作过滤条件，求取它的选择率。
- oprjoin：这个属性指定一个函数，把 OpExpr 当作连接条件，求取它的选择率。

每个操作符在 PG_OPERATER 系统表中都会指定对应的 oprrest 和 oprjoin，restriction_selectivity 函数就负责调用 OpExpr 表达式的操作符对应的 oprrest，而 join_selectivity 函数就负责调用 OpExpr 表达式的操作符对应的 oprjoin，如果没有指定 oprrest 或者 oprjoin，就使用默认的选择率 0.5。PG_OPERATOR 系统表中的每个操作符对应的 oprrest 属性和 oprjoin 属性的值如表 5-10 所示。

表 5-10　操作符与选择率的计算函数对照表

oprrest对应的函数名	oprjoin对应的函数名	操作符
eqsel	eqjoinsel	=
neqsel	neqjoinsel	<>
scalarltsel	Scalarltjoinsel	<= 、<
scalargtsel	scalargtjoinsel	>=、>
positionsel	positionjoinsel	<<、&<、&>、>>
contsel	contjoinsel	<@、@>
areasel	areajoinsel	>=、>、<、<=、?#
regexeqsel	regexeqjoinsel	~
regexnesel	regexnejoinsel	!~
likesel	likejoinsel	~~
nlikesel	nlikejoinsel	!~~
networksel	networkjoinsel	<<、>>、>>=、&&
icnlikesel	icnlikejoinsel	!~~*
iclikesel	iclikejoinsel	~~*
arraycontsel	arraycontjoinsel	&&、@>、<@
tsmatchsel	tsmatchjoinsel	@@、@@@
rangesel	contjoinsel	@>、<@

我们不打算一一分析这些函数，但是其中比较重要的几个函数，我们有必要占用一些篇幅来分析一下。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

5.3.1　eqsel 函数

eqsel 函数和 neqsel 函数的实现方式相同，因为：

$$neqsel = 1 - eqsel - nullfrac;$$

因此 eqsel 函数和 neqsel 函数都调用了 eqsel_internal 函数来计算选择率，在 eqsel_internal 函数中通过 negate 参数来区分是 eqsel 调用的还是 neqsel 调用的，如图 5-10 所示。

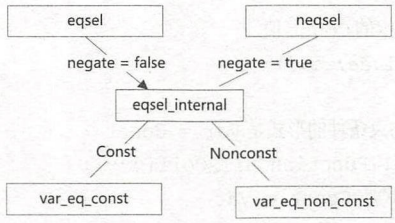


图 5-10　等值条件的选择率计算

在 eqsel_internal 函数中如果发现 negate == true，那么就代表是 neqsel 函数调用的。例如对于约束条件是 a <> 5，会在 eqsel_internal 函数中将“<>”操作符转换成“=”操作符，这样可以获得 a = 5 的选择率，然后就可以反向计算 a<>5 的选择率。

```
//如果是 negate == true，就代表是利用 neqsel 函数求选择率
if (negate)
{
    //查找当前操作符的反义操作符，'<>'的反义操作符是'='
    operator = get_negator(operator);
    if (!OidIsValid(operator))
    {
        //如果没有反义符号，使用默认的选择率
        return 1.0 - DEFAULT_EQ_SEL;
    }
}
```

如果约束条件是 Var = Const 或者 Const = Var 的形式，那么可以调用 var_eq_const 函数来估算它的选择率。

在 var_eq_const 函数中，如果发现约束条件中引用的列属性具有唯一性质，那么等值操作所获得的选择率一定是（1/元组的数量）。例如 STUDENT 表中的 sno 列上有主键索引，那么 sno 列就不会出现重复的值，对于约束条件 sno = 1 而言，查询只能获得 1 条元组作为结果，所以选择率取决于 STUDENT 表中元组的数量。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

如果约束条件中的列上有统计信息存在，而且约束条件中的常量（Const）出现在统计信息的高频值数组中，那么就直接从高频值数组中获得选择率。

```
//从统计信息中获取高频值的数组
if (get_attstatsslot(&sslot, vardata->statsTuple,
                STATISTIC_KIND_MCV, InvalidOid,
                ATTSTATSSLOT_VALUES | ATTSTATSSLOT_NUMBERS))
{
   ......
   //遍历高频值数组，匹配约束条件中的 Const 值
   for (i = 0; i < sslot.nvalues; i++)
   {
      if (varonleft)  //如果约束条件的形式是 Var = Const
         match = DatumGetBool(FunctionCall2Coll(……);
      else   //如果约束条件的形式是 Const = Var
         match = DatumGetBool(FunctionCall2Coll(……);
      if (match)//匹配成功之后，跳出循环
         break;
   }
}

if (match)
{
   //匹配成功，就获得对应高频值中记录的选择率
   selec = sslot.numbers[i];
}
```

如果约束条件中的 Const 没有在统计信息中匹配到高频值，那么这时候可以先将 NULL 值和高频值对应的比例去掉，让除掉 NULL 值和高频值之外的值平均分享剩余的选择率。

```
//把所有 MCV 中的比例累加起来
for (i = 0; i < sslot.nnumbers; i++)
   sumcommon += sslot.numbers[i];
//除了 NULL 值和高频值之外，还剩下多少比例
selec = 1.0 - sumcommon - nullfrac;

//修正这个比例
CLAMP_PROBABILITY(selec);

//对剩余的值消重，获得共有多少个数据
otherdistinct = get_variable_numdistinct(vardata, &isdefault) -
   sslot.nnumbers;
```

版权所有，严禁传播，违者自负法律责任！


```
//剩余的值共享剩余的比例
if (otherdistinct > 1)
   selec /= otherdistinct;

//既然 Const 没有出现在高频值数组中，选择率不能比高频值中最低的比例高
if (sslot.nnumbers > 0 && selec > sslot.numbers[sslot.nnumbers - 1])
   selec = sslot.numbers[sslot.nnumbers - 1];
```

var_eq_non_const 函数的处理过程和 var_eq_const 函数略有不同，主要是因为 var_eq_non_const 函数处理的约束条件中没有常量值，所以难以从统计信息中获取有价值的信息，估算的成分更大一些。

```
//因为约束条件中没有常量，因此无法使用高频值数组（MCV）
//这里假设除 NULL 值后，其余的值分布是平坦的
selec = 1.0 - nullfrac;
ndistinct = get_variable_numdistinct(vardata, &isdefault);
if (ndistinct > 1)
   selec /= ndistinct;

//根据上述假设获得的选择率，不能比高频值数组中的比例高
if (get_attstatsslot(&sslot, vardata->statsTuple,
                  STATISTIC_KIND_MCV, InvalidOid,
                  ATTSTATSSLOT_NUMBERS))
{
//如果平均选择率比高频值数组中最小的选择率还要高，那么转换成高频值数组中最小的选择率
   if (sslot.nnumbers > 0 && selec > sslot.numbers[0])
       selec = sslot.numbers[0];
   free_attstatsslot(&sslot);
}
```

5.3.2　scalargtsel 函数

如果约束条件中是'>'、'>='、'<= '、'<'这些操作符，那就需要通过 scalargtsel 函数或者 scalarltsel 函数来获得选择率，在这两个函数中，对约束条件要进行调整，保证 Var 出现在约束条件的左侧，如果 Var 出现在右侧，则需要调换它们的顺序，并且通过 get_commutator 函数获得反向操作符，调换顺序之后，scalargtsel 函数和 scalarltsel 函数都调用了 scalarineqsel 函数。

如果约束条件中的 Var（也就是对应的列属性）上没有统计信息，那么选择率默认是 DEFAULT_INEQ_SEL。

版权所有，严禁传播，违者自负法律责任！

如果统计信息中有高频值数组，那么需要统计两个值，一个是高频值数组中符合约束条件的高频值所占的比例（mcv_selec），另一个是所有的高频值数组中所有的值占的总的比例（sumcommon）。例如统计信息中有高频值{1, 2, 3, 4, 5}，每个高频值的比例为{0.10, 0.09, 0.08, 0.07, 0.06}，如果要计算约束条件 a > 3 的选择率，可以先计算 mcv_selec = 0.06 + 0.07 = 0.13 和 sumcommon = 0.10 + 0.09 + 0.08 + 0.07 + 0.06 = 0.40。

如果统计信息中有直方图，那么需要统计约束条件在直方图中所占的比例，因为统计信息中的直方图是等频直方图，首先使用二分法找到 Const 值所在的桶。

```
//用二分法找到约束条件中的 Const 所在的桶
while (lobound < hibound)
{
   int       probe = (lobound + hibound) / 2;
   bool      ltcmp;

   //增加 get_actual_variable_range 函数是尝试解决这样一个问题：
   //在统计信息生成后，对于更新比较频繁的表，可能会写入大量的值小于直方图的
   //下界或者高于直方图的上界，在不更新统计信息的情况下，
   //如果约束条件中的 Const 也小于直方图的下界或者高于直方图的上界
   //选择率的计算就会不准确，进而影响查询的代价的准确性，因此：
   //如果遇到这种情况，而且表上有索引，则可以通过索引获得当前列的最大值或最小值，
   //将最大值或者最小值更新到直方图的上界或者下界，这样估计就会准确一些。
   //这样校正上界或者下界带来的问题是：替换了边界值的桶实际的行数变多了，
   //直方图不再是一个等频直方图
   if (probe == 0 && sslot.nvalues > 2)
      have_end = get_actual_variable_range(root,
                                           vardata,
                                           sslot.staop,
                                           &sslot.values[0],
                                           NULL);
   else if (probe == sslot.nvalues - 1 && sslot.nvalues > 2)
      have_end = get_actual_variable_range(root,
                                           vardata,
                                           sslot.staop,
                                           NULL,
                                           &sslot.values[probe]);

   ltcmp = DatumGetBool(FunctionCall2Coll(opproc,
                                          DEFAULT_COLLATION_OID,
                                          sslot.values[probe],
```

版权所有，严禁传播，违者自负法律责任！


```
                                    constval));
    ......
}
```

在确定了约束条件中的 Const 值所在的桶之后，就确定了“小于当前桶”的所有桶中的值是满足约束条件的，但是一个桶中可能包含多个值，因此这个 Const 值在“当前桶”中所占的比例仍然需要计算。

```
if (lobound <= 0)
{
   //Const 小于直方图的最小边界值，也就是直方图中的所有值都不满足约束条件
   histfrac = 0.0;
}
else if (lobound >= sslot.nvalues)
{
   //Const 大于直方图的最小边界值，也就是直方图中的所有值都满足约束条件
   histfrac = 1.0;
}
else
{
   ......
   //如果 Const 命中一个桶，还需要看 Const 在桶内的比例
   if (convert_to_scalar(constval, consttype, &val,
                         sslot.values[i - 1], sslot.values[i],
                         vardata->vartype,
                         &low, &high))
   {

      if (high <= low) //桶的边界值相等，则假设 Const 在桶内占比 0.5
         binfrac = 0.5;
      else if (val <= low) //Const 值是桶的下界，也就是桶内没有满足约束条件的值
         binfrac = 0.0;
      else if (val >= high) //Const 值是桶的上界，也就是桶内的所有值都满足约束条件
         binfrac = 1.0;
      else
      {
         //假设桶内的数据分布是线性的，这样就可以用线性的方法来获得桶内的选择率
         binfrac = (val - low) / (high - low);

         //如果桶内的选择率是个无意义的值，那么设置为默认值 0.5
         if (isnan(binfrac) ||
            binfrac < 0.0 || binfrac > 1.0)
```

```
            binfrac = 0.5;
        }
    }
    else
    {
        //直接设定桶内的默认值是0.5
        binfrac = 0.5;
    }

    //先获得直方图中有几个桶的值满足约束条件
    histfrac = (double) (i - 1) + binfrac;
    //因为是等频直方图，所以（满足约束条件的桶数/总桶数） = 约束条件在直方图中的选择率
    histfrac /= (double) (sslot.nvalues - 1);
}
```

经过计算之后，获得的 histfrac 是 Const 值在直方图中占的选择率，由于直方图中的值并不是列属性上的全部值，因此还需要变换直方图的选择率，变换后直方图中满足约束条件的值所占的总比例为（1.0 – NULL 值比例–高频值总比例）×histfrac 。

```
//stats->stanullfrac:NULL值比例
//sumcommon:所有高频值所占的比例之和
//直方图中的所有值占属性（列）中所有值的比例
selec = 1.0 - stats->stanullfrac - sumcommon;

//hist_selec:直方图中满足约束条件的值占直方图中所有值的比例
if (hist_selec >= 0.0)
    selec *= hist_selec;//获得直方图中满足约束条件的值占属性（列）中所有值的比例
else
    selec *= 0.5;  //默认值0.5

//再加上高频值数组中满足约束条件的值的比例，即为最终选择率
selec += mcv_selec;
```

5.3.3 eqjoinsel 函数

对过滤条件（这里指的是含有常量值的条件）的选择率估计要容易一些，因为其中有一个常量值，这时统计信息就能发挥非常大的作用。无论是 MCV 统计信息还是直方图统计信息，都能结合常量值进行计算，只要生成统计信息是数据采样获得的样本是显著的，那么相对而言选择率的估计就是准确的。

但连接条件（含有常量值的连接条件在选择率的估算中按照过滤条件计算）中由于不含有常量值，操作符两端都是变量，在使用统计信息时就无从下手。对于非等值的连接条件（如 Var > Var、Var < Var 等），PostgreSQL 数据库直接使用了默认值 DEFAULT_INEQ_SEL，而对于等值的连接条件，选择率的计算也需要建立一些假设。在 PostgreSQL 数据库非常古老的版本中，就假设了连接条件引用的两个表的数据是平坦的，这样就能够估算出对于 LHS 的每一个值对应的 RHS 可以产生的连接结果的数量。

例如有一个连接条件 A.a = B.b，A 表的全部数据量是 *Na*，消重之后的数据量是 *NDa*，B 表的数据量是 *Nb*，消重之后的数据量是 *NDb*。

那么对于 A 表的 a 属性的每一个值，和 B 表的 b 属性做等值连接，平均都会产生 *Nb*/*NDb* 个结果，再精确一点，因为等值连接的操作符是严格的，因此我们可以把 NULL 值去掉，假设 B 表的空值率是 *nullfrac_b*，那么对于 A 表的 a 属性的每一个值，和 B 表的 b 属性做等值连接，都会产生 *Nb*×（1.0 - *nullfrac_b*）/*NDb* 个结果。

如果 A 表的空值率是 *nullfrac_a*，那么 A 中的非 NULL 值的数量是 *Na*×（1.0 – *nullfrac_a*），那么可以估计出连接的结果共有：

$$Na \times (1.0 - nullfrac\_a) \times Nb \times (1.0 - nullfrac\_b) / NDb$$

A 表和 B 表在没有连接条件的情况下可以产生连接结果的数量是：

$$Na \times Nb$$

因此可以计算出连接条件 A.a = B.b 的选择率是：

$$Na \times (1.0 - nullfrac\_a) \times Nb \times (1.0 - nullfrac\_b) / NDb / (Na \times Nb)$$
$$= (1.0 - nullfrac\_a) \times (1.0 - nullfrac\_b) / NDb$$

假设连接条件交换顺序，变成了 B.b = A.a，依照上述的计算方式可以获得选择率是：

$$(1.0 - nullfrac\_b) \times (1.0 - nullfrac\_a) / NDa$$

对于这种情况，PostgreSQL 数据库采取了折中的手段，最终使用的选择率是：

$$(1.0 - nullfrac\_a) \times (1.0 - nullfrac\_b) / \mathrm{MAX}(NDa, NDb)$$

但是受 Y. Ioannidis 和 S.Christodoulakis 的文献 *On the propagation of errors in the size of join results* 的启发，PostgreSQL 数据库如果发现操作符两端的属性（列）的统计信息中都有高频值数组，那么对高频值单独处理会获得更准确的选择率。

这样 PostgreSQL 在数据库对等值连接的估算过程中，将选择率分成了如下 4 部分，如图 5-11 所示。

- NULL 值率部分：一个表的某一个列属性的 NULL 值的比例。
- 高频值数组中匹配互相部分：约束条件中两个表的列都有高频值统计信息，那么两个高频值数组取交集，也就可以获得相互匹配的选择率。
- 高频值数组中未匹配的部分：去掉高频值数组中互相匹配的值，剩余值的选择率。
- 其他值的部分：去掉 NULL 值、去掉高频值数组中的值，其他的值的数量。

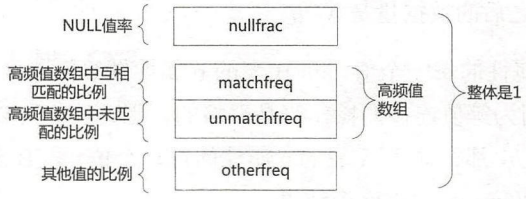


图 5-11　选择率分成 4 个部分分别计算

如图 5-11 所示的高频值数组值互相匹配的选择率的获取，需要对两个高频值数组做时间复杂度是 $O(n^2)$的遍历，将两个高频值数组中共有的相同值的选择率统计出来。

```
//用一个复杂度是 O(n^2) 的循环来查找两个属性的统计信息中的高频值数组是否有相等的值
for (i = 0; i < sslot1.nvalues; i++)
{
   int          j;
   for (j = 0; j < sslot2.nvalues; j++)
   {
      if (hasmatch2[j]) //用一个 hasmatch2 数组记录 RHS 的表中匹配的 MCV 项
         continue;
      if (DatumGetBool(FunctionCall2Coll(&eqproc,
                                 DEFAULT_COLLATION_OID,
                                 sslot1.values[i],
                                 sslot2.values[j])))
      {
         hasmatch1[i] = hasmatch2[j] = true; //找到匹配项
         //记录匹配项产生的选择率，匹配项的选择率乘积就是真正的选择率
         matchprodfreq += sslot1.numbers[i] * sslot2.numbers[j];
         nmatches++;
         break;
      }
   }
```

```
}

//计算 LHS 的表的列属性的匹配比例和没有匹配上的比例
//RHS 的计算方式和 LHS 的计算方式相同
matchfreq1 = unmatchfreq1 = 0.0;
for (i = 0; i < sslot1.nvalues; i++)
{
   if (hasmatch1[i]) //匹配到的 MCV 项的比例记录到 matchfreq
      matchfreq1 += sslot1.numbers[i];
   else  //没有匹配上的 MCV 项的比例记录到 unmatchfreq
      unmatchfreq1 += sslot1.numbers[i];
}
```

如图 5-11 所示的 NULL 值率（nullfrac）可以直接使用来自统计信息中的 stanullfrac，那么其他值的比例（otherfreq）就是（1.0 – nullfrac – matchfreq – unmatchfreq）。

连接条件的选择率可以这样计算，假如 A.a 共有 *Na* 行数据，B.b 共有 *Nb* 行数据，同时在 A 表中有 *S*(*Na*)行可以和 B 表产生等值连接，在 B 表中有 *S*(*Nb*)行可以产生等值连接，那么连接条件 A.a = B.b 的选择率就应该是：

$$selec. = \frac{S(Na)}{Na} \times \frac{S(Nb)}{Nb}$$

$$= \frac{S(x1+x2+x3+x4)}{Na} \times \frac{S(y1+y2+y3+y4)}{Nb}$$

$$= \frac{S(x1)+S(x2)+S(x3)+S(x4)}{Na} \times \frac{S(y1)+S(y2)+S(y3)+S(y4)}{Nb}$$

$$= \frac{S(x1)}{Na} \times \frac{S(y1)}{Nb} + \frac{S(x1)}{Na} \times \frac{S(y2)}{Nb} + \frac{S(x1)}{Na} \times \frac{S(y3)}{Nb} + \frac{S(x1)}{Na} \times \frac{S(y4)}{Nb} + \frac{S(x2)}{Na} \times \frac{S(y1)}{Nb} + \frac{S(x2)}{Na} \times \frac{S(y2)}{Nb} +$$
$$\frac{S(x2)}{Na} \times \frac{S(y3)}{Nb} + \frac{S(x2)}{Na} \times \frac{S(y4)}{Nb} + \frac{S(x3)}{Na} \times \frac{S(y1)}{Nb} + \frac{S(x3)}{Na} \times \frac{S(y2)}{Nb} + \frac{S(x3)}{Na} \times \frac{S(y3)}{Nb} + \frac{S(x4)}{Na} \times \frac{S(y4)}{Nb} +$$
$$\frac{S(x4)}{Na} \times \frac{S(y1)}{Nb} + \frac{S(x4)}{Na} \times \frac{S(y2)}{Nb} + \frac{S(x4)}{Na} \times \frac{S(y3)}{Nb} + \frac{S(x4)}{Na} \times \frac{S(y4)}{Nb}$$

其中：

$$Na = x1 + x2 + x3 + x4$$
$$Nb = y1 + y2 + y3 + y4$$

假设：

- $\frac{S(x1)}{Na}$是 A.a 的“NULL 值率”，$\frac{S(y1)}{Nb}$是 B.b 的“NULL 值率”。
- $\frac{S(x2)}{Na}$是 A.a 的“高频值数组中互相匹配值比例”，$\frac{S(y2)}{Nb}$是 B.b 的“高频值数组中互相匹配值比例”。

- $\frac{S(x3)}{Na}$是 A.a 的“高频值数组中未匹配值比例”，$\frac{S(y3)}{Nb}$是 B.b 的“高频值数组中未匹配值比例”。
- $\frac{S(x4)}{Na}$是 A.a 的“其他值的比例”，$\frac{S(y4)}{Nb}$是 B.b 的“其他值的比例”。

那么$\frac{S(x1)}{Na} \times \frac{S(y1)}{Nb}$就代 A.a 的 NULL 值部分的值和 B.b 的 NULL 值的部分进行连接产生的选择率，这个选择率只是这两部分进行连接产生的连接结果的选择率，也就是说我们把连接条件的选择率的求值转换成了 4 个部分依次求选择率(共 16 个子选择率)，最终汇总成总的选择率。

既然$\frac{S(x1)}{Na}$对应的是 A.a 的 NULL 值部分的能产生连接结果的比例，而这部分里面的值应该全部是 NULL 值，又因为等值操作的操作符是严格的，所以任何其他 3 个部分和 NULL 值进行连接，都不会产生连接结果，同理，$\frac{S(y1)}{Nb}$所代表的 B.b 的 NULL 值部分可能产生的连接结果的比例也是 0。因此只要算式中出现了$\frac{S(x1)}{Na}$或$\frac{S(y1)}{Nb}$，这部分的选择率就是 0。例如对于$\frac{S(x4)}{Na} \times \frac{S(y1)}{Nb}$而言，它代表的是 A.a 的其他值和 B.b 的 NULL 值进行连接产生的选择率，因为操作符是严格的，所以$\frac{S(x4)}{Na} \times \frac{S(y1)}{Nb}$等于 0，因此可以去掉和$\frac{S(x1)}{Na}$和$\frac{S(y1)}{Nb}$相关的 7 个子算式，然后开始讨论剩下的 9 个算式。

这里我们给出一个例子，如表 5-11 所示。结合这个例子，对余下算式的选择率进行分析，如表 5-12 所示。

表 5-11　选择率 4 个部分对应的示例

属性	空值部分	MCV匹配的部分	MCV未匹配的部分	其他值部分
A.a	{NULL}	{2，5}	{3，7}	{4，8，9}
B.b	{NULL}	{2，5}	{4，6，8}	{1，7，9}

表 5-12　选择率的 4 个部分衍生出的 9 个算式说明

算式	含义	选择率
$\frac{S(x2)}{Na} \times \frac{S(y2)}{Nb}$	A.a的高频值数组相互匹配值部分和B.b的高频值数组相互匹配值部分进行连接产生的选择率	高频值数组相互匹配的部分做连接，因为是完全相同的项，所以都能产生连接结果，例如2在A.a中占得总比例是10%，2在B.b中占的总比例是20%，那么约束条件在2这个值上的选择率就是10% ×20% = 2%

续表

算式	含义	选择率
$\frac{S(x2)}{Na} \times \frac{S(y3)}{Nb}$	A.a的高频值数组相互匹配的部分和B.b的高频值数组未匹配的部分进行连接产生的选择率	这两部分做等值连接不会产生连接结果，因为在匹配的时候，A.a的高频值数组相互匹配的部分和B.b的高频值数组未匹配的部分是没有匹配上的，所以选择率是0
$\frac{S(x2)}{Na} \times \frac{S(y4)}{Nb}$	A.a的高频值数组相互匹配的部分和B.b的其他值的部分进行连接产生的选择率	这两部分做等值连接不会产生连接结果，我们反证一下：A.a的高频值数组相互匹配的部分和B.b的其他值的部分如果能产生连接结果，那么在B.b中这个值就应该出现在B.b的高频值数组相互匹配的部分，也就是说，示例中在B.b的其他值的部分不可能出现{2，5}中的任何一个值，所以选择率是0
$\frac{S(x3)}{Na} \times \frac{S(y2)}{Nb}$	A.a的高频值数组未匹配的部分和B.b的高频值数组相互匹配的部分进行连接产生的选择率	这两部分做等值连接不会产生连接结果，和其他值的部分原因相似，所以选择率是0
$\frac{S(x3)}{Na} \times \frac{S(y3)}{Nb}$	A.a的高频值数组未匹配的部分和B.b的高频值数组未匹配的部分进行连接产生的选择率	都是为匹配上的值，肯定不能产生连接结果，所以选择率是0
$\frac{S(x3)}{Na} \times \frac{S(y4)}{Nb}$	A.a的高频值数组未匹配的部分和B.b的其他值的部分进行连接产生的选择率	应用(1.0 - nullfrac_a) ×(1.0 - nullfrac_b)/ MAX(NDa，NDb) 对这部分单独求解
$\frac{S(x4)}{Na} \times \frac{S(y2)}{Nb}$	A.a的其他值的部分和B.b的高频值数组相互匹配的部分进行连接产生的选择率	这两部分做等值连接不会产生连接结果，和 $\frac{S(x2)}{Na} \times \frac{S(y4)}{Nb}$ 原因相似，所以选择率是0
$\frac{S(x4)}{Na} \times \frac{S(y3)}{Nb}$	A.a的其他值的部分和B.b的高频值数组未匹配的部分进行连接产生的选择率	应用(1.0 - nullfrac_a) ×(1.0 - nullfrac_b)/ MAX(NDa，NDb) 对这部分单独求解
$\frac{S(x4)}{Na} \times \frac{S(y4)}{Nb}$	A.a的其他值的部分和B.b的其他值的部分进行连接产生的选择率	应用(1.0 - nullfrac_a) ×(1.0 - nullfrac_b)/ MAX(NDa，NDb) 对这部分单独求解

综合上面的分析，最终可以得到连接条件的选择率是：

$$selec. = \frac{S(x2)}{Na} \times \frac{S(y2)}{Nb} + \frac{S(x3)}{Na} \times \frac{S(y4)}{Nb} + \frac{S(x4)}{Na} \times \frac{S(y3)}{Nb} + \frac{S(x4)}{Na} \times \frac{S(y4)}{Nb}$$

$$= \frac{S(x2)}{Na} \times \frac{S(y2)}{Nb} + \frac{S(x3)}{Na} \times \frac{S(y4)}{Nb} + \frac{S(x4)}{Na} \times \left(\frac{S(y3)}{Nb} + \frac{S(y4)}{Nb}\right)$$

选择率的计算过程对应到源代码中的情况如下：

```
// (S(x2)/Na × S(y2)/Nb) 部分的计算，在匹配的同时做了计算
matchprodfreq += sslot1.numbers[i] * sslot2.numbers[j];

totalsel1 = matchprodfreq;
if (nd2 > sslot2.nvalues)
//S(x3)/Na × S(y4)/Nb 应用 (1.0 - nullfrac_a) × (1.0 - nullfrac_b) / MAX (NDa, NDb )
   totalsel1 += unmatchfreq1 * otherfreq2 / (nd2 - sslot2.nvalues);
if (nd2 > nmatches)
//S(x4)/Na × (S(y3)/Nb + S(y4)/Nb)应用 (1.0 - nullfrac_a) × (1.0 - nullfrac_b) / MAX (NDa, NDb )
   totalsel1 += otherfreq1 * (otherfreq2 + unmatchfreq2) / (nd2 - nmatches);
```

5.4 小结

本章分成了两个部分，一部分是介绍统计信息，另一部分是介绍选择率，其中统计信息是选择率计算的基础。

PostgreSQL 数据库在 10.0 之前的版本只有单列的统计信息，在 10.0 版本中增加了基于多列的统计信息，但是目前多列统计信息的内容还比较单薄，这可能是 PostgreSQL 数据库需要继续改进的地方。

如果约束条件的选择率估算能够使用统计信息，估算的结果相对还是准确的，它出现偏差的主要问题是需要经常假设数据的分布是平坦的，这种假设并不是经常性地成立。如果约束条件无法使用统计信息进行选择率的计算，那么约束条件的选择率的计算会遇到“严重”的困难，PostgreSQL 数据库通常会采用默认值来处理这种情况，这种处理会导致代价的估算出现偏差，但是目前仍然没有有效的手段来对这种情况进行改进。

6

第 6 章
扫描路径

扫描路径是对基表进行遍历时的执行路径，针对不同的基表有不同的扫描路径，例如针对堆表有顺序扫描（SeqScan）、针对索引有索引扫描（IndexScan）和位图扫描（BitmapScan）、针对子查询有子查询扫描（SubqueryScan）、针对通用表达式有通用表达式扫描（CteScan）等，扫描路径通常是执行计划树的叶子节点。如图 6-1 所示是创建扫描路径的函数调用关系。

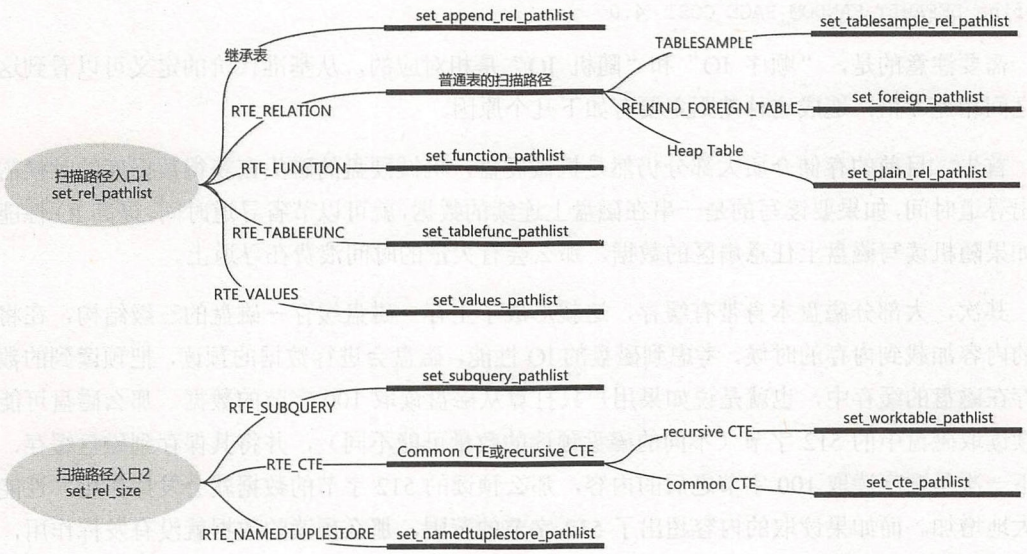


图 6-1　扫描路径创建函数调用关系

6.1 代价（Cost）

我们已经介绍了约束条件的选择率，也就是知道了通过扫描路径要扫描出来的结果所占的比例或者通过连接操作所获得的元组所占的比例，通过这个比例就可以推算出中间结果和最终结果的数量，进而使用这些数量来计算代价。

6.1.1 代价基准单位

在实际应用中，数据库用户的硬件环境千差万别，如 CPU 的频率、主存的大小和磁盘介质等因素都会影响执行计划的实际执行效率，因此在代价估算的过程中，我们无法获得“绝对真实”的代价。而且“绝对真实”的代价也是不必要的，因为我们只是想从多个路径（Path）中找到一个代价最小的路径，只要这些路径的代价是可以“相互比较”的就可以了。因此可以设定一个“相对”的代价作为单位 1，同一个查询中所有的物理路径都基于这个“相对”的单位 1 来计算代价，这样计算出来的代价就是可以比较的，也就能用来对路径进行挑选了。

6.1.1.1 基于页面的 IO 基准代价

PostgreSQL 数据库采用顺序读写一个页面的 IO 代价作为单位 1，用 DEFAULT_SEQ_PAGE_COST 来表示。

```
#define DEFAULT_SEQ_PAGE_COST  1.0
#define DEFAULT_RANDOM_PAGE_COST  4.0
```

需要注意的是，“顺序 IO”和“随机 IO”是相对应的，从基准代价的定义可以看到这二者之间相差 4 倍，造成这种差距主要有如下几个原因。

首先，目前的存储介质大部分仍然是机械硬盘，机械硬盘的磁头在获得数据库的时候需要付出寻道时间，如果要读写的是一串在磁盘上连续的数据，就可以节省寻道时间，提高 IO 性能，而如果随机读写磁盘上任意扇区的数据，那么会有大量的时间浪费在寻道上。

其次，大部分磁盘本身带有缓存，这就形成了主存→磁盘缓存→磁盘的三级结构，在将磁盘的内容加载到内存的时候，考虑到磁盘的 IO 性能，磁盘会进行数据的预读，把预读到的数据保存在磁盘的缓存中，也就是说如果用户只打算从磁盘读取 100 字节的数据，那么磁盘可能会连续读取磁盘中的 512 字节（不同的磁盘预读的数量可能不同），并将其保存到磁盘缓存，如果下一次是顺序读取 100 字节之后的内容，那么预读的 512 字节的数据就会发挥作用，性能会大大地增加。而如果读取的内容超出了 512 字节的范围，那么预读的数据就没有发挥作用，磁盘的 IO 性能就会下降。

寻道时间和预读是否命中对查询的性能影响还是非常大的，最明显的对比就是顺序扫描（SeqScan）和索引扫描（IndexScan）这两种路径，SeqScan 是对数据从头至尾地遍历。由于 PostgreSQL 数据库的表的数据以堆的方式存储，因此可以假设这个表的数据在磁盘上是连续的，因此顺序扫描的 IO 代价是 DEFAULT_SEQ_PAGE_COST。而索引扫描则不然，以 B 树索引为例，B 树上的叶子节点保存了表中每个元组的 ItemId，我们在索引扫描时是对 B 树的叶子节点进行顺序扫描，但是我们每获得一个叶子节点，都要去读取叶子节点中的 ItemId 对应的元组，这个操作是一个随机 IO 操作，因此索引扫描在代价计算的时候需要考虑 DEFAULT_RANDOM_PAGE_COST。

顺序 IO 和随机 IO 的值并非一成不变，例如固态硬盘正在逐步取代机械硬盘，那么顺序 IO 的代价和随机 IO 的代价是否还有 4 倍的差距就值得商榷了，这就需要数据库的使用者根据硬件环境来灵活地调整这些基础的代价值。为了能够让用户可以自己配置这些值，PostgreSQL 数据库定义了对应的 GUC 参数来代表这些值。

```
double seq_page_cost = DEFAULT_SEQ_PAGE_COST;
double random_page_cost = DEFAULT_RANDOM_PAGE_COST;
```

同时，由于不同的数据文件存储在不同的磁盘介质上，PostgreSQL 数据库允许用户在创建表空间的时候指定顺序 IO 和随机 IO 的基准代价。例如，有些数据文件存储在机械硬盘上，而另一些数据文件存储在固态硬盘上，它们的 IO 性能是不一样的，因此用户可以设定基准代价如下：

```
CREATE TABLESPACE TEST_SPC LOCATION '...' WITH (SEQ_PAGE_COST=2, RANDOM_PAGE_COST=3);
```

在计算代价的时候，如果表空间上指定了对应的单位代价，就按照已经指定的单位计算，如果没有指定，则按照默认的单位代价计算。

6.1.1.2　基于元组的 CPU 基准代价

在讨论了基于页面的 IO 代价之后，我们来看一下基于元组的 CPU 代价。读取页面通常并不是查询的终极目标，查询的终极目标是将元组以要求的形式展示出来，因此就产生了从页面读取元组以及对元组处理的代价，这部分代价不同于读取页面的 IO 代价，这时页面已经在主存中了，从主存中的页面获取元组不会产生磁盘 IO，它的代价主要产生在 CPU 的计算上。

PostgreSQL 数据库定义 DEFAULT_CPU_TUPLE_COST 来表示处理一条元组的代价，使用 DEFAULT_CPU_INDEX_TUPLE_COST 来表示处理一条索引元组的代价。

```
#define DEFAULT_CPU_TUPLE_COST      0.01
#define DEFAULT_CPU_INDEX_TUPLE_COST 0.005
```

如果用户要调整这两个基准代价，那么可以调整 cpu_tuple_cost 和 cpu_index_tuple_cost 两

个变量。

```
double cpu_tuple_cost = DEFAULT_CPU_TUPLE_COST;
double cpu_index_tuple_cost = DEFAULT_CPU_INDEX_TUPLE_COST;
```

6.1.1.3 基于表达式的 CPU 基准代价

在执行计划的过程中，不止处理元组需要消耗 CPU 资源，在投影、约束条件中包含大量的表达式，对这些表达式求值同样需要消耗 CPU 资源，因此 PostgreSQL 数据库把表达式的求值代价单独剥离出来。使用 DEFAULT_CPU_OPERATOR_COST 来作为计算表达式代价的基准单位，用户可以通过调整 cpu_operator_cost 来调整这个基准单位。

```
#define DEFAULT_CPU_OPERATOR_COST  0.0025
double cpu_operator_cost = DEFAULT_CPU_OPERATOR_COST;
```

6.1.1.4 并行查询产生的基准代价

目前，PostgreSQL 数据库部分支持了并行查询，因此通常在分布式数据库系统中才考虑的通信代价目前 PostgreSQL 数据库也需要考虑了，因为 Gather 进程和 Worker 进程在并行查询的过程中需要进行通信，因此需要考虑进程间通信（IPC）所需的初始化代价（DEFAULT_PARALLEL_SETUP_COST），以及 Worker 进程向 Gather 进程投递元组的代价（DEFAULT_PARALLEL_TUPLE_COST）。

```
#define DEFAULT_PARALLEL_TUPLE_COST 0.1
#define DEFAULT_PARALLEL_SETUP_COST  1000.0
```

PostgreSQL 数据库也为这些变量定义了对应的参数，用户可以调整这些参数来适应自己当前的情况。

```
double parallel_tuple_cost = DEFAULT_PARALLEL_TUPLE_COST;
double parallel_setup_cost = DEFAULT_PARALLEL_SETUP_COST;
```

6.1.1.5 缓存对代价的影响

数据库本身有缓存系统，磁盘上也有磁盘缓存，当读取一个缓存中的数据页面时是不会产生磁盘 IO 的，因此，如果对每个页面都计算磁盘 IO 的代价，代价的计算结果就会失真，所以我们还需要对缓存中的页面数量有一个估计，目前 PostgreSQL 数据库用 effective_cache_size 参数来表示，实际上这个值一定是不准确的，这是 PostgreSQL 数据库需要改进的地方。

```
#define DEFAULT_EFFECTIVE_CACHE_SIZE  524288
int effective_cache_size = DEFAULT_EFFECTIVE_CACHE_SIZE;
```

6.1.2　启动代价和整体代价

PostgreSQL 数据库将代价分成了两个部分：启动代价（Startup Cost）和执行代价（Run Cost），两者的和是整体代价（Total Cost）。

Total Cost = Startup Cost + Run Cost

在 Path 结构体中用 startup_cost 和 total_cost 两个变量来表示启动代价和整体代价，startup_cost 是指从语句开始执行到查询引擎返回第一条元组的代价（另一种说法是准备好获得第一条元组的代价），total_cost 是 SQL 语句从开始执行到结束的所有代价。

我们知道查询优化器会对不同的路径进行代价对比，筛选代价比较低的路径作为执行路径，那么为什么还要区分 startup_cost 和 total_cost？下面看一个示例。

```
postgres=# CREATE INDEX TEST_A_A_IDX ON TEST_A(a);
CREATE INDEX
postgres=# INSERT INTO TEST_A SELECT GENERATE_SERIES(1,10000),FLOOR(RANDOM()*100),
FLOOR(RANDOM()*100), FLOOR(RANDOM()*100);
INSERT 0 10000
postgres=# ANALYZE TEST_A;
ANALYZE
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE a > 1 ORDER BY a;
                        QUERY PLAN
-------------------------------------------------------------------------
 Sort  (cost=844.39..869.39 rows=10000 width=16)
   Sort Key: a
   ->  Seq Scan on test_a  (cost=0.00..180.00 rows=10000 width=16)
         Filter: (a > 1)
(4 rows)
```

从示例 SQL 语句可以看出，SQL 语句要求查询的结果有序（ORDER BY a），我们已知属性 a 上有 B 树索引，而 B 树索引是有序的，所以 IndexScan 是一个好选择，但是从示例的查询计划可以看出它没有选择 IndexScan，反而选择了 SeqScan+Sort 的方式，这是由于 a > 1 的选择率比较高，导致 IndexScan 产生的随机读比较多，页面的 IO 代价中随机 IO 的代价远高于顺序 IO 的代价，因此没有选择 IndexScan。

```
--禁用 SeqScan
postgres=# SET ENABLE_SEQSCAN=FALSE;
SET
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE a > 1 ORDER BY A;
                           QUERY PLAN
```

```
------------------------------------------------------------------------------
 Index Scan using test_a_a_idx on test_a  (cost=0.29..7373.28 rows=10000 width=16)
   Index Cond: (a > 1)
(2 rows)
```

从示例的执行计划可以看出，禁用 SeqScan 扫描之后，执行计划变成了对 B 树索引进行扫描，索引扫描的代价是 7373.28，而 SeqScan+Sort 方式的总代价是 869.39，所以我们说查询优化器的选择是正确的。同时还注意到，SeqScan+Sort 方式的启动代价是 844.39，而 IndexScan 的启动代价是 0.29，也就是说 SeqScan+Sort 方式的启动代价高于 IndexScan。

如果给 SQL 语句增加 LIMIT 子句就会出现一个问题，LIMIT 子句是 Non-SPJ 操作，它处在执行计划的上层（可以参考第 9 章中的内容），也就是说下层路径是不知道上层有没有进行 LIMIT 限制的，对于如下 SQL 语句：

```
SELECT * FROM TEST_A WHERE a > 1 ORDER BY a LIMIT 1;
```

假如查询优化器仍然会选择 SeqScan+Sort 这样的启动代价比较高的路径作为子路径，然后形成 SeqScan+Sort+Limit 这样的路径，这就不合理了。因为由于子句 LIMIT 1 的出现，之前放弃的 IndexScan 就有了优势，虽然 IndexScan 会出现大量的随机 IO，但是 LIMIT 1 子句限制了随机 IO 的数量，IndexScan 不用再扫描整个索引了，只需要从索引获取一条元组，也就是说这时候随机 IO 只有一次，因此 IndexScan 这种启动代价比较低的路径的优势就凸显出来了。下面来分别看一下加上 Limit 子句之后，两种执行计划的表现。

```
postgres=# SET ENABLE_SEQSCAN=TRUE;
SET
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE a > 1 ORDER BY a LIMIT 1;
                              QUERY PLAN
------------------------------------------------------------------------
 Limit  (cost=0.29..0.32 rows=1 width=16)
   ->  Index Scan using test_a_a_idx on test_a
         Index Cond: (a > 1)
(3 rows)

postgres=# SET ENABLE_INDEXSCAN=FALSE;
SET
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE a > 1 ORDER BY a LIMIT 1;
                         QUERY PLAN
------------------------------------------------------------------------
 Limit  (cost=230.00..230.00 rows=1 width=16)
   ->  Sort  (cost=230.00..255.00 rows=10000 width=16)
```

```
        Sort Key: a
        ->  Seq Scan on test_a  (cost=0.00..180.00 rows=10000 width=16)
              Filter: (a > 1)
(5 rows)
```

从示例中可以看出，加上了 LIMIT 1 子句之后，查询优化器选择了启动代价比较低的 IndexScan，IndexScan 的启动代价是 0.29，而 SeqScan+ Sort 的启动代价是 230.00（注意，如果没有 LIMIT 1 子句，SeqScan+ Sort 的启动代价是 844.39，这个示例的启动代价是 230.00 的原因是：查询优化器在计算代价的时候对 LIMIT 1 这种情况做了优化，对于带有 LIMIT 子句的语句，它可能会采用基于 TOP-K 的堆排序，这样就能降低排序所带来的启动代价）。

6.1.3　表达式代价的计算

我们已经知道表达式代价的基准单位是 cpu_operator_cost，不同的表达式需要辅以基准单位进行计算，表达式代价主要包括如下方面：

- 对投影列的表达式进行计算产生的代价。
- 对约束条件中的表达式进行计算产生的代价。
- 对函数参数中的表达式进行计算产生的代价。
- 对聚集函数中的表达式进行计算产生的代价。
- 子计划等执行计算产生的代价。

表达式代价的计算是通过 cost_qual_eval 函数（针对表达式列表）或 cost_qual_eval_node 函数（针对单个表达式）来计算的，这两个函数的作用本质上没什么区别，略有不同的是 cost_qual_eval_node 函数可以对 List 列表进行递归处理（只是调用栈的深度增加了一层）。cost_qual_eval 函数和 cost_qual_eval_node 函数都调用了递归函数 cost_qual_eval_walker，cost_qual_eval_walker 函数递归处理表达式并且将表达式的估计代价逐层累加到 QualCost 结构体中，QualCost 结构体的定义如下：

```
typedef struct QualCost
{
    Cost      startup;      /* 计入启动代价的部分 */
    Cost      per_tuple;    /* 应用到元组上的代价 */
} QualCost;
```

cost_qual_eval_walker 函数对各种表达式的处理如表 6-1 所示。

表 6-1　表达式代价的计算方法

表达式类型	计算方法
FuncExpr、OpExpr、DistinctExpr、NullIfExpr	目前在PG_PROC系统表中对每个函数都给予了procost属性，FuncExpr、OpExpr、DistinctExpr、NullIfExpr等表达式的代价计算方法是： per_tuple = procost × cpu_operator_cost
ScalarArrayOpExpr	ScalarArrayOpExpr表达式可能无须将ARRAY中的数值判定完成即可获得表达式的最终值，因此ScalarArrayOpExpr表达式的代价计算方法是： per_tuple = procost × cpu_operator_cost × sizeof(ARRAY) × 0.5 其中0.5是硬性指定的 示例：… WHERE A > ANY(ARRAY[1,2,3]);
Aggref、WindowFunc	Aggref、WindowFunc不在这里计算表达式代价，它们都和聚集函数相关，在生成聚集路径的时候计算代价

CoerceViaIO	CoerceViaIO针对一些特定的类型，它需要计算输出和输入两个操作符的代价
	示例：
	`CREATE TABLE TEST_JSON(a JSON);`
	`INSERT INTO TEST_JSON VALUES('{"Name":"Tom"}');`
	`SELECT a::JSONB FROM TEST_JSON;`

ArrayCoerceExpr	对数组类型的变量进行类型转换，需要对数组中的每一个变量都进行类型转换，因此该表达式的代价计算方法是：
	per_tuple = procost × cpu_operator_cost × sizeof(ARRAY)
	示例：
	`CREATE TABLE TEST_ARRCO(a INT[]);`
	`INSERT INTO TEST_ARRCO VALUES(ARRAY[1,2,3]);`
	`SELECT a::INT8[] FROM TEST_ARRCO;`

RowCompareExpr	ROW类型的比较次数由ROW类型中的元素个数决定，因此该表达式的代价计算方法是：
	`per_tuple = procost1 × cpu_operator_cost + procost2 × cpu_operator_cost + … + procostN × cpu_operator_cost`
	示例：
	`select row(1,'aaa') > row(2,'bbb');`

续表

表达式类型	计算方法

CurrentOfExpr	Current Of语法目前借助TID扫描实现，因此在这里加大它的代价，也就是在它原有代价的基础上再增加disable_cost，在生成TID路径的时候再将disable_cost减下去，借助这种方式Current Of语法最终就会选择TID扫描：
	startup += disable_cost
	示例：
	`CREATE TABLE TEST_CUROF(a INT);`
	`INSERT INTO TEST_CUROF VALUES(1);`
	`BEGIN;`
	`DECLARE C1 CURSOR FOR SELECT * FROM TEST_CUROF`
	`FETCH 1 FROM C1;`
	`DELETE FROM TEST_CUROF WHERE CURRENT OF C1;`
	`COMMIT;`

SubPlan	分别累加subplan的启动代价和执行代价，subplan->per_call_cost是subplan执行的代价，因为在表达式中的subplan会随着表达式的调用而多次执行
	计算方法：
	`startup += subplan->startup_cost;`
	`per_tuple += subplan->per_call_cost;`

AlternativeSubPlan	目前只有一种使用AlternativeSubPlan的情况，就是在为EXISTS类型的子连接生成子计划的时候，可能会生成待选的多个子计划，在对执行器执行子计划的时候从待选的子计划中选择一个最优的。 当前还不知道选择哪个子计划作为执行计划，硬性选择第一个子计划计算代价
PlaceHolderVar	这里不计算该表达式的代价，该表达式的代价在set_rel_width函数或add_placeholders_to_joinrel函数中计算
MinMaxExpr、SQLValueFunction、XmlExpr、CoerceToDomain、NextValueExpr	这些表达式的计算代价硬性指定为cpu_operator_cost

6.2 路径（Path）

物理优化对查询树的改造与逻辑优化不同，逻辑优化主要是基于 SQL 语句中指定的逻辑运算符进行等价的变换，而物理优化则在逻辑优化的基础之上建立一棵路径树，路径树中的每一个路径都是一个物理算子，这些物理算子是为查询执行器准备的，查询执行器通过运行这些物理算子来实现查询中的逻辑运算，这些物理算子又大体可以分为两类，它们是扫描路径和连接路径。

所谓的扫描路径是针对基表而言的，这些基表就是 RELOPT_BASEREL 类型的 RelOptInfo，我们认为这些基表是一个二维的关系实体，扫描路径就是对这个二维的关系实体进行遍历的过程，它包括顺序扫描路径、（快速）索引扫描路径、位图扫描路径、TID 扫描路径等。

而所谓的连接路径则是记录基表之间的物理连接关系，我们已经将表之间的逻辑连接关系记录到了 SpecialJoinInfo 结构体中，在物理优化的阶段会根据这些逻辑连接关系建立一个新的 RelOptInfo，然后将基于逻辑连接关系建立的物理连接路径记录到这个 RelOptInfo 中，并且边记录边筛选。例如针对 InnerJoin 这样一个逻辑连接关系，查询优化器可以建立 NestloopJoin、HashJoin 等物理连接路径，这些物理连接路径都可以实现 InnerJoin 的运算，但是它们的路径代价是不同的，因此物理优化的一个主要工作就是建立物理连接路径并且选出代价最低的物理连接路径。

需要注意的是，在筛选物理路径的时候，并不是只选择一个整体代价最低的路径就可以了，而是要记录多种类型的路径，比如需要记录整体代价最低的路径、启动代价最低的路径、代价比较低的参数化路径等，记录多个物理路径的原因可以参考启动代价的说明。

6.2.1 Path 结构体

在 PostgreSQL 数据库中，路径使用 Path 结构体来表示，Path 结构体“派生”自 Node 结构体，Path 结构体同时也是一个“基”结构体，类似于 C++中的基类，每个具体路径都从 Path 结构体中“派生”，例如索引扫描路径使用的 IndexPath 结构体就是从 Path 结构体中“派生”的。

```
typedef struct Path
{
    NodeTag       type;
    NodeTag       pathtype;          /* 路径的类型，可以是 T_IndexPath、T_NestPath 等 */
    RelOptInfo *parent;              /* 当前路径服务于哪个逻辑连接操作对应的 RelOptInfo */
    PathTarget *pathtarget;          /* 路径的投影，也会保存表达式代价*/
    ParamPathInfo *param_info;       /* 执行期参数，在执行器中，子查询或者一些特殊*/
```


非卖品，仅供非商业用途或交流学习使用

```
                                    /* 类型的连接需要实时地获得另一个表的当前值 */

    bool     parallel_aware;        /* 并行参数，区分并行还是非并行 */
    bool     parallel_safe;         /* 并行参数，由 set_rel_consider_parallel 函数决定 */
    int      parallel_workers;      /* 并行参数，并行进程的数量 */

    double   rows;                  /* 当前路径执行产生的中间结果估计有多少数据 */
    Cost     startup_cost;          /* 启动代价，从语句执行到获得第一条结果的代价 */
    Cost     total_cost;            /* 当前路径的整体执行代价 */

    List   *pathkeys;               /* 当前路径产生的中间结果的排序键值，如果无序则为 NULL */
} Path;
```

6.2.2　并行参数

变量 parallel_workers 代表的是并行度，所谓并行度就是对于一个任务同时需要几个并行的后台线程进行处理，用户可以在创建表的时候指定 parallel_workers 参数，例如通过 SQL 语句 CREATE TABLE TEST_A(a INT) WITH (PARALLEL_WORKERS=100)就可以指定一个并行度是 100 的表，但这并不代表对这个表进行扫描的时候一定会产生一个并行度是 100 的并行扫描路径，一方面非并行的路径代价可能低于并行路径的代价，这时就会选择非并行路径，另一方面 PostgreSQL 数据库对并行度也进行了限制，每个查询都有过大的并行度对数据库的整体性能也会带来不利的影响。

PostgreSQL 数据库通过 compute_parallel_worker 函数来获得实际可用的并行度，这个函数的参数是即将尝试生成并行路径的基表、对基表进行顺序扫描所需要的页面数（heap_pages）、对基表进行索引扫描所需要的页面数（index_pages），截至本书成稿，对基表进行索引扫描所需的页面数全部默认值是-1，代表不需要考虑这个参数。

PostgreSQL 数据库还增加了几个 GUC 参数来提高并行度设置的灵活性，这样用户就可以自己调节这些参数，根据当前的硬件环境来配置自己的并行度，这几个 GUC 参数如表 6-2 所示。

表 6-2　并行扫描参数说明

参数名	参数类型	描述
min_parallel_table_scan_size	int	启用并行表扫描的最小页面数，也就是说表扫描的页面数小于min_parallel_table_scan_size，不会启用并行扫描

非卖品，仅供非商业用途或交流学习使用

续表

参数名	参数类型	描述
min_parallel_index_scan_size	int	启用并行索引扫描的最小页面数，也就是说索引扫描的页面数小于min_parallel_index_scan_size，不会启用并行扫描
max_parallel_workers_per_gather	int	每个gather下的最大并行度，在同一个执行计划里，gather路径是并行的最上层子路径，它用来对并行的后台线程的执行结果进行合并，一个执行计划里可能有多个gather路径

对于基表的扫描，如果 heap_pages < min_parallel_table_scan_size 或者 index_pages < min_parallel_index_scan_size，就不启用并行查询。

另外基于 min_parallel_table_scan_size 和 min_parallel_index_scan_size 可以获得一个并行度的参考值。

$$min(\log_3 \frac{\text{heap\_pages}}{\text{min\_parallel\_table\_scan\_size}},\ \log_3 \frac{\text{index\_pages}}{\text{min\_parallel\_index\_scan\_size}})$$

当然，这个并行度还不能超过 max_parallel_workers_per_gather 的值，如果超过了 max_parallel_workers_per_gather，那么就取 max_parallel_workers_per_gather 的值作为并行度。

变量 parallel_safe 主要结合了子路径的 parallel_safe 及路径中的 RelOptInfo->consider_parallel 的值来判断当前的路径能否是并行安全的，并不是所有的查询路径都能应用并行查询。例如对于含有 PARAM_EXEC 类型的 Param 参数的查询路径，就不能并行。另外，对于 Gather 路径，由于 PostgreSQL 数据库的并行查询不能嵌套执行，因此直接设定为非并行是安全的。

变量 parallel_aware 则用来标识一个表是否真正应用了并行，它最早在 Path 中通过并行度来进行识别，如果并行度大于 0，就代表这个路径是并行路径，如果这个并行路径最终被查询优化器选定为执行路径，那么这个路径就可以生成并行执行计划（Plan），变量 parallel_aware 会被同时传递给并行执行计划（Plan）。在 Gather 路径下的子路径不一定都是并行路径，例如对于并行 HashJoin（实际上是 parallel-oblivious hash join 类型的并行 HashJoin，详情可以参考连接路径中关于 hash_inner_and_outer 函数的介绍），这里可以给出这样的一个示例（如果要生成下面示例中的执行计划，TEST_A 表中的数据量要多一些，另外可能需要尝试关闭一些其他连接方式，例如 SET ENABLE_NESTLOOP = FALSE）。

非卖品，仅供非商业用途或交流学习使用

```
postgres=# EXPLAIN SELECT * FROM TEST_A, TEST_B WHERE TEST_A.a = TEST_B.a;
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Gather  (cost=1001.02..8134.36 rows=64 width=32)
   Workers Planned: 2
   ->  Hash Join  (cost=1.02..7127.96 rows=27 width=32)
         Hash Cond: (test_a.a = test_b.a)
         ->  Parallel Seq Scan on test_a  (cost=0.00..6126.67 rows=266667 width=16)
         ->  Hash  (cost=1.01..1.01 rows=1 width=16)
               ->  Seq Scan on test_b  (cost=0.00..1.01 rows=1 width=16)
(7 rows)
```

从示例的查询计划可以看出，GatherPath 下有两个表：TEST_B 表作为 Hash Join 的内表，为每个并行的后台进程建立一个哈希表，TEST_A 表作为 Hash Join 的外表，每个后台的并行进程可以获得它的一部分数据，Hash Join 就是把外表的部分数据和基于内表建立 Hash 表做探测，这时虽然 TEST_A 表和 TEST_B 表都在 Gather 路径下的表，但是它们并非都需要并行，因此 TEST_A 表对应的 parallel_aware=true，而 TEST_B 表对应的 parallel_aware=false。

6.2.3　参数化路径

在谓词下推的过程中，对于…FROM A JOIN B ON A.a = B.b 这种类型的约束条件肯定是不能下推的，因为 A.a = B.b 这样的约束条件既引用了 LHS 表的列属性，又引用了 RHS 表的列属性，必须在获得两个表的元组之后才能应用这样的约束条件。现在假如在 B.b 属性上有一个索引 B_b_index，但如果没有能够匹配索引的约束条件，索引扫描路径或者不会被采纳，或者采纳为对整个索引进行扫描（例如 Fast Index Scan，这时候将索引视为一个表，实际上和 SeqScan 类似），发挥不了索引对数据筛选的作用。

再假设一下…FROM A JOIN B ON A.a = B.b 生成的执行计划是 Nestloop Join，A 表作为外表做 SeqScan，B 表作为内表也做 SeqScan。

```
Nestloop Join (JoinCluase: A.a = B.b)
    ->SeqScan (A)
    ->SeqScan (B)
```

它的执行过程应该是这样的：

1）从 A 表取出一条元组。

2）如果 A 表已经扫描完毕，执行结束。

非卖品，仅供非商业用途或交流学习使用

3）从 B 表取出一条元组。

4）如果 B 表已经扫描完毕，跳转到步骤 1）。

5）对两个元组应用 A.a = B.b 的约束条件。

6）如果符合 A.a = B.b 的约束条件，则返回连接结果，下一次执行跳转到步骤 3）。

7）如果不符合 A.a = B.b 的约束条件，则直接跳转到步骤 3）。

通过这些执行步骤可以看出，步骤 1）从 A 表取出一条元组之后，还需要扫描整个 B 表来匹配 A.a = B.b 这样的约束条件，而实际上在步骤 1）获得了 A 表的元组之后，A.a 的值已经确定了，假如我们把这个值从 A 表的元组中取出来作为一个常量，这时候约束条件就可以转换成 B.b = Const（交换 Const = B.b 的顺序），而这种只引用了连接操作一端的表的约束条件，大部分情况下是能够下推的，如果在 B.b 上有索引，那么 B.b = Const 这样的约束条件让索引变得有用了，索引扫描可以避免扫描整个 B 表。这时候执行计划就变成了：

```
Nestloop Join
    ->SeqScan (A) (Extract A.a to Const)
    ->IndexScan (B_b_index) (Filter: B.b = Const)
```

使用 B.b = Const 这样的约束条件对索引进行扫描，扫描的效率远高于 SeqScan，显然这种方法是可取的，但随之带来的问题是：如何将从 A 元组中提取出来的常量值传递到 B 的扫描中呢？PostgreSQL 数据库选择了通过参数的方式进行传递，也就是生成参数化的路径，这种方法的流程如图 6-2 所示。

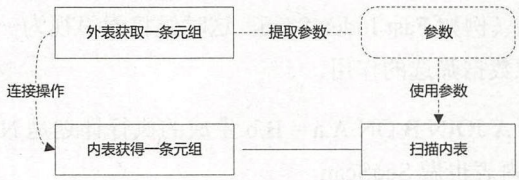


图 6-2　参数化路径说明

PostgreSQL 数据库已经定义了 Param 结构体，它就是负责传递参数的，这时候我们是否能利用这个结构体为我们传递参数呢？通常，Param 用来从执行计划外向执行计划内传递参数，比如 PBE（Prepare、Bind、Execute）中使用了 Param，它是从执行计划外向执行计划内 Bind 参数，再比如父执行计划和子执行计划之间通过 Param 进行“通信”，可以是从父执行计划传递给子执行计划，也可以从子执行计划传递给父执行计划，这种情况参数值也来自于执行计划外，而目前要用的参数在执行计划内传递，从一个路径向另一个路径传递参数值，或者说从外

非卖品，仅供非商业用途或交流学习使用

表向内表传递参数，和上面使用 Param 的方式有些区别，因此 PostgreSQL 数据库在这里定义了 NestLoopParam 结构体来封装 Param，保证查询执行器在执行 Nested Loop Join 执行算子的时候，能够根据 NestLoopParam 找到对应的 Param 进行参数传递，以实现这种执行计划内传递参数的功能。

```
typedef struct NestLoopParam
{
    NodeTag       type;
    int           paramno;          /* number of the PARAM_EXEC Param to set */
    Var           *paramval;        /* outer-relation Var to assign to Param */
} NestLoopParam;
```

既然是从外表向内表传递参数，这就产生了一个限定条件，就是在生成执行路径的时候，如果要建立参数化路径，就需要考虑谁是内表或者谁是外表的问题，产生参数的一方必须是外表，而使用参数的一方就必须是内表，因此在生成连接路径的时候必须先获知连接的双方中谁是参数的生产者，谁是参数的使用者，生产者在“前”，使用者在“后”。

除了考虑内外表连接顺序的问题，还需要考虑另一个问题，常用的物理连接路径有 Nestloop Join、Hash Join 和 Merge Join，它们是否都适合进行参数的传递？目前来看答案是否定的，PostgreSQL 数据库目前只支持了 Nestloop Join 下的内外表进行参数传递，这是由于只有 Nestloop Join 的两个表之间具有“驱动”关系，外表是“驱动表”，内表是“被驱动表”，而 Hash Join 和 Merge Join 则不具有这样的驱动关系，比如 HashJoin 就无法从外表的元组上提取出值并且应用到内表的执行路径上。

下面我们看一个参数化路径的例子。

```
postgres=# EXPLAIN SELECT * FROM TEST_A, TEST_B WHERE TEST_A.a = TEST_B.a AND TEST_A.a > 9000;
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Nested Loop  (cost=0.29..152931.52 rows=1 width=32)
   ->  Seq Scan on test_a  (cost=0.00..152889.96 rows=5 width=16)
         Filter: (a > 9000)
   ->  Index Scan using test_b_a_idx on test_b  (cost=0.29..8.30 rows=1 width=16)
         Index Cond: (a = test_a.a)
(5 rows)
```

从例子中可以看出，对 TEST_B 表使用了索引扫描，约束条件 TEST_A.a = TEST_B.a 下推到了索引扫描上，虽然示例的执行计划中显式的约束条件是 a = test_a.a，但实际上在执行计划中这里的 test_a.a 是一个 Param。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

6.2.4 PathKey

排序操作是数据库中常见的操作，而这个操作又是比较耗时的，如果在一个查询中需要对同一数据多次排序，那么可以考虑它们之间是否能共用一次的排序结果，从而避免多次排序的代价，下面看这样一个例子。

```
postgres=# EXPLAIN SELECT a FROM TEST_A ORDER BY a;
                              QUERY PLAN
-----------------------------------------------------------------------------------------
 Index Only Scan using test_a_a_idx on test_a  (cost=0.29..270.29 rows=10000 width=4)
(1 row)

postgres=# SET ENABLE_INDEXSCAN=FALSE;
SET
postgres=# EXPLAIN SELECT a FROM TEST_A ORDER BY a;
                        QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=153035.39..153060.39 rows=10000 width=4)
   Sort Key: a
   ->  Seq Scan on test_a  (cost=0.00..152371.00 rows=10000 width=4)
(3 rows)
```

从示例可以看出，同样的 SQL 语句执行了不同的扫描路径，分别是 IndexOnlyScan 和 SeqScan，其中 IndexOnlyScan 没有进行 Sort 操作，而 SeqScan 做了 Sort 操作，原因显而易见，B 树索引本身是有序的，对索引进行扫描之后产生的结果也是有序的，对有序的结果可以省掉 ORDER BY a 带来的 Sort 操作。

我们再来看一个示例。

```
postgres=# CREATE TABLE TEST_A(a INT, b INT);
postgres=# CREATE TABLE TEST_A(a INT, b INT);

postgres=# EXPLAIN SELECT * FROM TEST_A, TEST_B WHERE TEST_A.a = TEST_B.a ORDER BY TEST_A.a;
                        QUERY PLAN
-----------------------------------------------------------------------
 Merge Join  (cost=317.01..711.38 rows=25538 width=16)
   Merge Cond: (test_a.a = test_b.a)
   ->  Sort  (cost=158.51..164.16 rows=2260 width=8)
         Sort Key: test_a.a
         ->  Seq Scan on test_a  (cost=0.00..32.60 rows=2260 width=8)
   ->  Sort  (cost=158.51..164.16 rows=2260 width=8)
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
        Sort Key: test_b.a
        ->  Seq Scan on test_b  (cost=0.00..32.60 rows=2260 width=8)
(8 rows)
```

这次的两个表 TEST_A 和 TEST_B 都没有索引，通过查询计划可以看出，虽然我们的 ORDER BY 子句是在最上层的，但是显而易见它是“下推”给了每个表，这样就选择了 MergeJoin 连接操作，保证了输出结果也是有序的，那么 PostgreSQL 数据库是否有这样的“下推”操作呢？

PostgreSQL 查询优化器将查询路径的生成分成了两个层次：一个层次是 Non-SPJ 优化，它主要处理窗口函数、GROUP BY 子句、聚集函数、ORDER BY 子句、LIMIT 子句等；另一个层次是 SPJ 优化，它主要处理基于关系代数理论的投影（Project）、选择（Select）、连接（Join）。Non-SPJ 优化主要在 grouping_planner 函数中处理，而 SPJ 优化主要在 query_planner 函数中实现，Non-SPJ 优化是在 SPJ 优化产生的结果的基础上增加新的 Non-SPJ 路径，也就是说 Non-SPJ 路径是逐层叠加在 SPJ 优化的结果之上的。

PostgreSQL 数据库对 Non-SPJ 操作做的优化比较有限，但通过分析也不难发现还是会有一些细节上的优化。例如，如果在 Non-SPJ 优化过程中发现包含 ORDER BY 子句，且 query_planner 返回的执行路径恰好是按照 ORDER BY 的顺序返回的，那么就“完美”了，这个 ORDER BY 子句就可以被优化了。因此可以得出一个结论，如果 SPJ 优化阶段能够返回一个 Non-SPJ 优化“期望”有序的路径，那么就可能节省排序的代价。

因此 PostgreSQL 数据库通过分析 Non-SPJ 子句中的内容，获得它的期望顺序，并将其保存在 PlannerInfo->query_pathkeys 里，PlannerInfo->query_pathkeys 会在 SPJ 优化阶段生成执行路径的时候进行“暗示”，如果你能符合这个顺序最好了。但这个“暗示”并不是强制的，它最终是取决于代价的，假如节省掉排序的代价之后仍然高于其他路径的代价，查询优化器仍然会选择其他代价低的路径。

因此 PostgreSQL 数据库并没有显式地下推 ORDER BY 的这种逻辑，它无论是优先选择索引扫描还是优先选择 MergeJoin，都是基于 Non-SPJ 子句带来的“提示”，这种提示带来的好处是避免在 SPJ 操作生成执行路径（query_planner 函数）的时候将 Non-SPJ 操作期望的有序路径丢弃。

和 PlannerInfo->query_pathkeys 相对应，在 Path 结构体中也有一个 pathkeys 变量，这个变量代表的是当前路径的输出结果是以什么键值排序的。PlannerInfo->query_pathkeys 属于从上层向下层的提示，是 Non-SPJ 优化阶段对 SPJ 优化阶段生成的路径提出的要求，Path->pathkeys 则是从下层向上层的提示，它告诉上层的 SPJ 路径及 Non-SPJ 优化阶段：我是以这样顺序输出的，你可以基于这个输出顺序做一些优化。

版权所有，严禁传播，违者自负法律责任！

例如在 B 树索引扫描路径中，就会参照索引的键值生成((Path*)IndexPath)->pathkeys，它提示上层的路径“我是以这样的顺序输出结果的，你可以使用这个顺序进行优化”，这个提示仍然不是强制的，上一层路径如果恰好是 MergeJoin，那么它就无须在对其进行排序了（MergeJoin 需要对内外表排序），上一层路径如果是 HashJoin，那么((Path*)IndexPath)-> pathkeys 就不会被使用。

PlannerInfo->query_pathkeys 和 Plan->pathkeys 都是 List 指针类型的链表，链表中的每个节点都是一个 PathKey 结构体，下面来分析这个结构体。

```
typedef struct PathKey
{
   NodeTag        type;
   EquivalenceClass *pk_eclass;      //等价类
   Oid            pk_opfamily;       //排序的操作符族
   int            pk_strategy;       //升序还是降序
   bool           pk_nulls_first;    //NULL 值优先还是 NULL 值在后
} PathKey;
```

每个 PahtKey 结构体代表排序的一个键值，例如针对 ORDER BY A,B,C 产生的 PlannerInfo->query_pathkeys，第一排序键是 A，第二排序键是 B，第三排序键是 C。再例如索引 TEST_IDX(A,B,C)产生的索引扫描的 Plan->pathkeys，会参照索引键值的顺序生成第一排序键是 A，第二排序键是 B，第三排序键是 C。

在一个查询语句中，如果输出结果是按照一个键值排序的，那么和这个键值在一个等价类中的其他属性在输出结果上也一定是有序的。例如对于 SQL 语句 SELECT * FROM TEST_A, TEST_B WHERE TEST_A.a = TEST_B.a ORDER BY TEST_A.a，语句中的 TEST_A.a = TEST_B.a 可以产生一个等价类{TEST_A.a , TEST_B.a}，也就是说 TEST_A.a 和 TEST_B.a 是等价的，那么无论是以 ORDER BY TEST_A.a 的顺序输出结果，还是按照 ORDER BY TEST_B.a 的顺序输出结果，语句的执行结果都是一样的。

6.3 make_one_rel 函数

顾名思义，make_one_rel 函数就是“生成一个关系”，实际上就是创建一个 RelOptInfo 结构体，这个结构体可以用来表示基表，也可以用来表示连接操作，make_one_rel 的过程就是向 RelOptInfo 中增加物理路径的过程。目前还不考虑上层的 Non-SPJ 操作，例如 GROUP BY、ORDER BY 等（也不是完全不考虑，例如启动代价就是为上层的 LIMIT 操作设计的，但主要考

版权所有，严禁传播，违者自负法律责任！

虑的是基表之间的连接关系）。

RelOptInfo 通过层层叠加最终形成一个“路径树”，这个路径树的叶子节点一定是表示基表的 RelOptInfo，它里面保存的是扫描路径，上层节点是表示连接操作的 RelOptInfo（还可能有物化、排序等节点），它里面保存的是连接路径。make_one_rel 通过动态规划的方法或者遗传算法对扫描路径进行规划，并且根据 SpecialJoinInfo 中记录的逻辑连接关系将基表组合到一起，最终形成最上层的“一个 RelOptInfo”。

make_one_rel 函数主要分成两个阶段：生成扫描路径的阶段和生成连接路径的阶段，如图 6-3 所示，本章重点关注生成扫描路径的阶段。

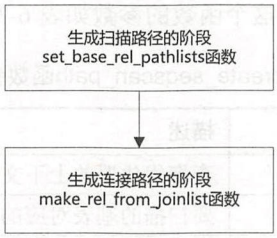


图 6-3　路径生成的两个阶段

6.4　普通表的扫描路径

在所有基表中最重要的、也是最常见的就是普通的堆表，它的扫描路径通过 set_plain_rel_pathlist 函数来设置，该函数对堆表尝试生成顺序扫描路径、并行顺序扫描路径、索引扫描路径和 TID 扫描路径，这些扫描路径需要通过 add_path 函数加入基表的 RelOptInfo->pathlist 中，如图 6-4 所示。

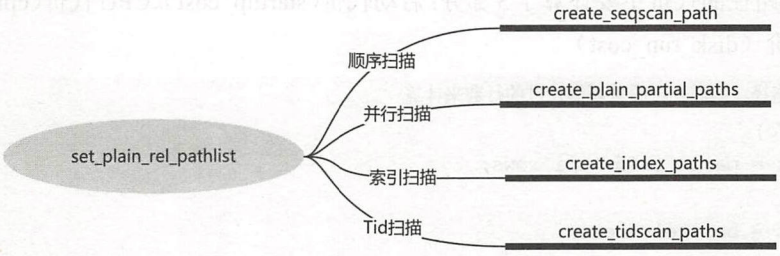


图 6-4　表上的扫描路径

版权所有，严禁传播，违者自负法律责任！

6.4.1 顺序扫描

顺序扫描（SeqScan）又称作全表扫描，是最基本的扫描方式，它的算法复杂度是 $O(N)$，其中 N 代表的是整个表中元组的数量，也就是说一个表中所有的元组都会被读取出来，读取出来的元组交付给约束条件进行过滤，把符合约束条件的元组交给投影函数进行投影，最终以查询语句指定的方式输出。

顺序扫描必需的要素有扫描对象（表）、过滤条件、投影列等，这些要素在 Path 结构体中都已经定义了，因此顺序扫描直接通过 Path 结构体来代表它的路径，没有单独地定义结构体（比如索引扫描就定义了单独的路径结构体 IndexPath）。顺序扫描的路径创建是在 create_seqscan_path 函数中完成的，这个函数的参数如表 6-3 所示。

表 6-3 create_seqscan_path函数参数说明

参数名	参数类型	描述
root	[IN] PlannerInfo *	查询优化器的上下文信息
rel	[IN] RelOptInfo *	要扫描的基表对应的结构体
required_outer	[IN] Relids	引用了哪些其他表的属性，大多数情况下就是RelOptInfo->lateral_relids
parallel_workers	[IN] int	通过compute_parallel_worker函数计算的并行度，需要注意的是，如果required_outer不为NULL，则parallel_workers必须为0

如果 required_outer 不为空，那么需要生成参数化的路径。目前，对于顺序扫描而言，只有 Lateral 变量出现在投影中时才会建立参数化路径，例如 SQL 语句 SELECT * FROM TEST_A LEFT JOIN LATERAL (SELECT a, TEST_A.a FROM TEST_B) bb ON TRUE，其中子查询的投影列中的 TEST_A.a 会被参数化。

顺序扫描路径的代价主要计算了 3 部分：启动代价（startup_cost）、CPU 代价（cpu_run_cost）、磁盘的 IO 代价（disk_run_cost）。

```
//如果是参数化路径，则需要按照重新估计过的行数来计算
if (param_info)
   path->rows = param_info->ppi_rows;
else
   path->rows = baserel->rows;

//如果表空间上指定了 SEQ_PAGE_COST 选项
//则按照表空间上指定的 SEQ_PAGE_COST 估计整体代价
```

版权所有，严禁传播，违者自负法律责任！


```
get_tablespace_page_costs(baserel->reltablespace, NULL,  &spc_seq_page_cost);

//SeqScan 是顺序读取，因此使用 SEQ_PAGE_COST
//SEQ_PAGE_COST × 页面数 = 页面读取的 IO 代价
disk_run_cost = spc_seq_page_cost * baserel->pages;

//获得扫描路径中对表达式求值的代价——也就是表达式代价
//这里获得的是约束条件中的表达式的代价
get_restriction_qual_cost(root, baserel, param_info, &qpqual_cost);

//在启动代价中增加约束条件表达式代价中的启动代价
startup_cost += qpqual_cost.startup;
//顺序扫描过程中需要处理每一条元组，并且在每一条元组上应用约束条件
cpu_per_tuple = cpu_tuple_cost + qpqual_cost.per_tuple;
//注意这里是 baserel->tuples，也就是表中所有的元组数量
cpu_run_cost = cpu_per_tuple * baserel->tuples;

//投影列的表达式代价累加，注意这里是 path->rows
//path->rows 是根据选择率估算出来的结果集的元组数
//因为并不是对表中所有的元组都做投影，投影操作只针对符合约束条件的元组
startup_cost += path->pathtarget->cost.startup;
cpu_run_cost += path->pathtarget->cost.per_tuple * path->rows;

//如果是并行查询，则对代价进行调整
if (path->parallel_workers > 0)
{
   //主查询进程对并行 worker 进程的执行效率会有促进的作用
   //worker 进程越多，这种促进的作用越小
   //目前采用算式 1.0 - (0.3 * path->parallel_workers) 来衡量这种促进的作用
   double parallel_divisor = get_parallel_divisor(path);

   //并行之后，查询分派到不同的 worker 进程上，CPU 代价分摊到每个 worker 进程上
   cpu_run_cost /= parallel_divisor;

   //每个 worker 进程的输出结果集变小了，分摊到每个 worker 进程上
   path->rows = clamp_row_est(path->rows / parallel_divisor);

   //每个进程读取的页面数也变小了，但是考虑到并行的 SeqScan 仍然是顺序读取
   //而且 SEQ_PAGE_COST 中已经考虑了磁盘对页面预取的情况，因此这里分摊 IO 代价
}
```

版权所有，严禁传播，违者自负法律责任！


```
//获得路径的启动代价和整体代价
path->startup_cost = startup_cost;
path->total_cost = startup_cost + cpu_run_cost + disk_run_cost;
```

6.4.2 索引扫描

PostgreSQL 数据库的表是堆组织表，数据在堆页面没有顺序排列的要求，向堆组织表插入一条数据时，可以将其存放在新开辟的页面中，也可以存放在之前删除元组时空出来的空白空间上，写入的位置没有特定的要求。这种形式带来的好处是堆组织表的结构比较简单，带来的问题是无法使用一些常用的查找算法来查询数据，通常只能采用顺序扫描（SeqScan）的方法，也就是对整个堆进行扫描，即使 SQL 语句中包含像 a=1 这样的约束条件，顺序扫描也无法应用任何优化措施。

为了提高数据库的查询性能，PostgreSQL 数据库提供了多种索引类型提供给各种不同的查询需求。例如 B 树索引，它是一种基于磁盘的树状数据组织结构，能够将像 a=1 这样的约束条件数据查找的复杂度减低成为 $O(\text{Log}(N))$。再例如 Hash 索引，我们知道 Hash 方法经常会用于数据查找，类似 a=1 这样的约束条件进行 Hash 查找的复杂度近似为 $O(1)$，因此索引是效率提升的有力武器。

索引本身是一种空间换时间的方法，通过对堆组织结构的数据进行“预处理”，使之变成为我们期望的组织结构，并且将这种期望的组织结构和堆组织表中的数据建立映射关系，类似书籍的“目录”，建立“标题”和“页面”的映射关系，从而达到快速读取所需数据的效果。

目前，PostgreSQL 数据库支持的索引类型比较多，通过查询 PG_AM 系统表可以看出它支持的主要有如下几种方式：

```
postgres=# SELECT oid, * FROM PG_AM;
 oid  | amname  |  amhandler   | amtype
------+-------- +------------- +--------
  403 | btree   | bthandler    | i
  405 | hash    | hashhandler  | i
  783 | gist    | gisthandler  | i
 2742 | gin     | ginhandler   | i
 4000 | spgist  | spghandler   | i
 3580 | brin    | brinhandler  | i
(6 rows)
```

PostgreSQL 数据库为每种索引指定了固定的索引访问方法（Access Method），在早期的 PostgreSQL 数据库中，这些方法保存在 PG_AM 表中，当前最新版本的 PG_AM 系统表只保留

版权所有，严禁传播，违者自负法律责任！

了 xxxhandler 方法，这些方法负责根据不同的索引类型对索引的特性进行设置，我们以 B 树索引为例。

```
Datum bthandler(PG_FUNCTION_ARGS)
{
    IndexAmRoutine *amroutine = makeNode(IndexAmRoutine);

    //B 树索引支持 <、<=、=、>、>= 共 5 种比较策略
    amroutine->amstrategies = BTMaxStrategyNumber;
    amroutine->amsupport = BTNProcs;//索引支持几种排序（Sort）接口，目前 B 树支持 2 种
    amroutine->amcanorder = true;//B 树索引能够支持 ORDER BY 操作
    amroutine->amcanorderbyop = false;//目前只有 Gist 类型的索引支持这种情况
    amroutine->amcanbackward = true;//B 树索引是否支持反向扫描
    amroutine->amcanunique = true;//B 树索引是否支持唯一索引
    amroutine->amcanmulticol = true;//B 树索引是否支持多键值索引
    amroutine->amoptionalkey = true;//使用 B 树索引是否要求在第一个键值上有约束条件
    //B 树索引是否支持 SAOP（Search Array Operator），即支持 ScalarArrayOpExpr 表达式
    amroutine->amsearcharray = true;
    amroutine->amsearchnulls = true;//B 树索引是否允许约束条件中有 IS [NOT] NULL
    amroutine->amstorage = false;
    amroutine->amclusterable = true;
    amroutine->ampredlocks = true;
    amroutine->amcanparallel = true;
    amroutine->amkeytype = InvalidOid;

    amroutine->ambuild = btbuild;//建立 B 树索引的接口
    amroutine->ambuildempty = btbuildempty;//清空 B 树索引的接口
    amroutine->aminsert = btinsert;//向 B 树索引中插入一个新的索引项
    amroutine->ambulkdelete = btbulkdelete;//批量删除 B 树索引项
    amroutine->amvacuumcleanup = btvacuumcleanup;//对 B 树索引进行 Vacuum 操作
    amroutine->amcanreturn = btcanreturn;//B 树索引能否直接返回索引项
    amroutine->amcostestimate = btcostestimate;//B 树索引扫描的代价估计函数
    amroutine->amoptions = btoptions;//该接口可以分析 B 树索引对应的 reloptions
    amroutine->amproperty = btproperty;
    amroutine->amvalidate = btvalidate;//验证 B 树索引对应的操作符类
    amroutine->ambeginscan = btbeginscan;//准备索引扫描
    amroutine->amrescan = btrescan;//索引扫描对应的 rescan
    amroutine->amgettuple = btgettuple;//获取索引项
    amroutine->amgetbitmap = btgetbitmap;//支持位图索引扫描函数
    amroutine->amendscan = btendscan;//结束索引扫描函数
    amroutine->ammarkpos = btmarkpos;//记录当前的索引扫描的位置（Pos）
```

版权所有，严禁传播，违者自负法律责任！


```
    amroutine->amrestrpos = btrestrpos;//恢复索引扫描的位置，和 ammarkpos 相对应
    amroutine->amestimateparallelscan = btestimateparallelscan; //并行索引扫描代价估计函数
    amroutine->aminitparallelscan = btinitparallelscan; //并行索引扫描初始化函数
    amroutine->amparallelrescan = btparallelrescan; //并行索引扫描的 rescan 函数

    PG_RETURN_POINTER(amroutine);
}
```

每种类型的索引都会有自己的 IndexAmRoutine，根据自己的特点设置对应的索引属性和索引访问方法，例如，对 B 树索引而言，它对应的操作符族中的操作符是可以用来支持 ORDER By 操作的，因此它的 IndexAmRoutine->amcanorder 的值是 true，而 Hash 索引的 IndexAmRoutine->amcanorder 的值就是 false，从 Hash 的性质也可以看出，Hash 索引对应的操作符族中的操作符无法支持排序操作。

在创建索引的时候需要对多种数据类型的列进行操作，PostgreSQL 数据库为每种数据类型都定义了对应的操作符类，这些操作符类保存在系统表 PG_OPCLASS 中，例如在 INT 类型的列上创建 B 树索引所对应的操作符类是：

```
postgres=# SELECT opcmethod,opcname,opcfamily,opcintype,opcdefault,opckeytype FROM
PG_OPCLASS WHERE opcmethod=403 AND opcintype=23;
 opcmethod | opcname  | opcfamily | opcintype | opcdefault | opckeytype
-----------+----------+-----------+-----------+------------+------------
       403 | int4_ops |      1976 |        23 | t          |          0
(1 row)
```

我们注意到其中有一列 opcfamily 的值是 1976，这个 OID 的来源是系统表 PG_OPFAMILY，PG_OPFAMILY 系统表的作用是将在 PG_OPCLASS 中具有兼容性的类型集中到同一个“族”中，在 PG_OPCLASS 系统表中，opcfamily = 1976 的 B 树操作符类有：

```
postgres=# SELECT opcmethod,opcname,opcfamily,opcintype,opcdefault,opckeytype FROM
PG_OPCLASS WHERE opcmethod=403 AND opcfamily=1976;
 opcmethod | opcname  | opcfamily | opcintype | opcdefault | opckeytype
-----------+----------+-----------+-----------+------------+------------
       403 | int2_ops |      1976 |        21 | t          |          0
       403 | int4_ops |      1976 |        23 | t          |          0
       403 | int8_ops |      1976 |        20 | t          |          0
(3 rows)
```

也就是说 int2_ops、int4_ops、int8_ops 这三个操作符类都属于同一个操作符族 integet_ops。

```
postgres=# select oid,* from pg_opfamily where oid=1976;
 oid  | opfmethod |   opfname    | opfnamespace | opfowner
------+-----------+--------------+--------------+----------
 1976 |       403 | integer_ops |           11 |       10
(1 row)
```

在 PG_OPCLASS 系统表和 PG_OPFAMILY 系统表中都没有定义具体的操作方法(函数)，这个工作在 PG_AMOP 系统表和 PG_AMPROC 中完成。

```
postgres=# select amopfamily,amoplefttype,amoprighttype,amopstrategy,amopopr,amopmethod
from pg_amop where amopfamily = 1976 and amopmethod=403 and amoplefttype =23 and
amoprighttype=23;
 amopfamily | amoplefttype | amoprighttype | amopstrategy | amopopr | amopmethod
------------+--------------+---------------+--------------+---------+------------
       1976 |           23 |            23 |            1 |      97 |        403
       1976 |           23 |            23 |            2 |     523 |        403
       1976 |           23 |            23 |            3 |      96 |        403
       1976 |           23 |            23 |            4 |     525 |        403
       1976 |           23 |            23 |            5 |     521 |        403
(5 rows)
```

这里有 5 行，也就和 B 树的 IndexAmRoutine->amstrategies 对应上了，每个类型都需要支持 5 种比较策略，另外对应的 IndexAmRoutine->amsupport 也应该能对应上，B 树索引支持的两个函数在 PG_AMPROC 系统表中。

```
postgres=# select * from pg_amproc where amprocfamily=1976 and amproclefttype=23 and
amprocrighttype=23;
 amprocfamily | amproclefttype | amprocrighttype | amprocnum |     amproc
--------------+----------------+-----------------+-----------+-------------------
       1976   |           23   |           23    |         1 | btint4cmp
       1976   |           23   |           23    |         2 | btint4sortsupport
(2 rows)
```

有了这些操作符，用户就可以“使用”索引了，但用户没有直接访问这些索引的接口，例如虽然针对一个表创建了索引，但是在对表进行扫描的时候，查询优化器选择顺序扫描还是索引扫描，用户通常没有办法决定（有些数据库提供了 HINT 功能，PostgreSQL 数据库没有该功能），路径的选择是由数据库查询优化模块来完成的。

下面看一下索引扫描有哪些路径可供我们选择，假设有如下表：

```
CREATE TABLE TEST_A(a INT, b INT, c INT, d INT, e INT);
```

```
INSERT INTO TEST_A SELECT GENERATE_SERIES(1,10000), FLOOR(RANDOM()*100),
FLOOR(RANDOM()*100), FLOOR(RANDOM()*100), FLOOR(RANDOM()*100);
--在TEST_A的{a,b,c}三个列创建一个B树索引
CREATE INDEX TEST_A_ABC_IDX ON TEST_A(a, b, c);
```

然后对 TEST_A 表进行查询，查看执行计划。

```
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE a = 9000;
                          QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using test_a_abc_idx on test_a  (cost=0.29..8.30 rows=1 width=20)
   Index Cond: (a = 9000)
(2 rows)
```

通过执行计划可以看出，对表 TEST_A 的查询使用了索引扫描（Index Scan using test_a_abc_idx），这是最经典的索引扫描路径，因为 TEST_A_ABC_IDX 是一个 B 树索引，做等值查询的算法时间复杂度是 $O(\text{Log}(N))$，因此相对顺序扫描而言性能大大提升了，这种索引扫描实际上执行了两个步骤：首先在 B 树上查找 a=9000 的索引项，然后根据索引项记录的堆元组的“地址”找到 TEST_A 堆表中 a=9000 的元组。

既然索引项已经记录了{a,b,c }三个列的值，那么如果查询的投影列是{a,b,c }的子集，由于索引中的数据库和堆表的数据是一致的，因此就可以不用去 TEST_A 堆表上拿数据了，直接使用索引项中的值就可以了，这种扫描路径被称为快速索引扫描（Index Only Scan using test_a_abc_idx），也就是它省掉了去堆上获取元组的消耗代价。

```
postgres=# EXPLAIN SELECT a,b,c FROM TEST_A WHERE A = 9000;
                             QUERY PLAN
---------------------------------------------------------------------------------
 Index Only Scan using test_a_abc_idx on test_a  (cost=0.29..8.30 rows=1 width=12)
   Index Cond: (a = 9000)
(2 rows)
```

另外，我们再将约束条件从 a = 9000 修改为 a > 9000，则执行计划改变为 BitmapHeapScan + BitmapIndexScan 的方式。

```
postgres=# EXPLAIN SELECT a,b,c FROM TEST_A WHERE A > 9000;
                            QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on test_a  (cost=28.04..104.54 rows=1000 width=12)
   Recheck Cond: (a > 9000)
```


严禁网络传播本PDF，违者责任自负！

```
  ->  Bitmap Index Scan on test_a_abc_idx  (cost=0.00..27.79 rows=1000 width=0)
        Index Cond: (a > 9000)
(4 rows)
```

示例中的执行计划没有使用 IndexScan 的方式直接获取数据，显然，通过 IndexScan 的方式也能够获得所有的数据，下面分析 IndexScan 和 BitmapIndexScan 的区别。IndexScan 中有“随机”读，虽然读取索引项的时候是顺序读取的，但是从索引项向堆数据映射的过程是随机的，在计算 IO 代价时，随机读产生的代价会占很大的比重，因为随机读的代价基准单位是 RANDOM_PAGE_COST=4，远高于顺序读取的代价基准单位 SEQ_PAGE_COST=1。如果把这种随机读转换成顺序读，IO 代价就可以降低下来，这时候我们可以在索引项和堆数据之间再增加一个 Bitmap（位图），在索引扫描的过程中获得索引项之后并不急于去堆上读数据，而是将数据项在堆上的元组的“地址”保存在 Bitmap 中，在索引项扫描完毕之后，Bitmap 中的数据项地址就变成了有序的，然后借由有序的 Bitmap 来读取堆上的数据，这样就把随机读转换成了顺序读。

通常而言，**如果查询中的约束条件的选择率比较大，那么查询优化器会倾向于选择顺序扫描；如果约束条件的选择率非常小，那么查询优化器会倾向于选择索引扫描；如果约束条件的选择率介于二者之间（中等），那么很可能就会选择位图扫描。**

再假设 TEST_A 表上没有 TEST_A_ABC_IDX 索引，而是在 A、B、C 三个列属性上分别建立 TEST_A_A_IDX、TEST_A_B_IDX、TEST_A_C_IDX 三个索引。

```
DROP INDEX TEST_A_ABC_IDX;
CREATE INDEX TEST_A_A_IDX ON TEST_A(A);
CREATE INDEX TEST_A_B_IDX ON TEST_A(B);
CREATE INDEX TEST_A_C_IDX ON TEST_A(C);
```

再看一下下面 SQL 语句的查询计划。

```
postgres=# EXPLAIN SELECT a,b,c FROM TEST_A WHERE A > 9000 AND (B > 99 OR C > 99);
                              QUERY PLAN
-----------------------------------------------------------------------------------------
 Bitmap Heap Scan on test_a  (cost=30.01..70.32 rows=18 width=12)
   Recheck Cond: (((b > 99) OR (c > 99)) AND (a > 9000))
   ->  BitmapAnd  (cost=30.01..30.01 rows=18 width=0)
         ->  BitmapOr  (cost=9.97..9.97 rows=185 width=0)
               ->  Bitmap Index Scan on test_a_b_idx  (cost=0.00..4.97 rows=92 width=0)
                     Index Cond: (b > 99)
               ->  Bitmap Index Scan on test_a_c_idx  (cost=0.00..4.98 rows=93 width=0)
                     Index Cond: (c > 99)
```


严禁网络传播本PDF，违者责任自负！

```
        ->  Bitmap Index Scan on test_a_a_idx  (cost=0.00..19.79 rows=1000 width=0)
              Index Cond: (a > 9000)
(10 rows)
```

从上面的示例可以看出位图扫描的另一个优点，它能综合使用多个索引扫描的结果，通过BitmapAnd（取交集）和BitmapOr（取并集）的方式将多个索引扫描的结果整合到一起。

通过上面的分析我们可以看出，索引扫描通常支持的路径包括普通索引扫描、快速索引扫描、位图索引扫描。另外需要注意的是，还可能会生成参数化索引扫描路径。

6.4.2.1 约束条件的匹配

生成索引扫描路径的前提是索引必须能用得上，查询中出现的投影列的不同或约束条件的不同都会导致生成不同的扫描路径，因此，在生成索引扫描路径之前，需要先查看索引的键和约束条件是否匹配。

在引入参数化路径之前，只需要匹配 RelOptInfo->baserestrictinfo，也就是说只匹配那些简单的indexkey op const或者const op indexkey类型的子约束条件。但是在引入了参数化路径之后，Const的含义发生了扩展，它不再仅仅是个常量，还可能是个“参数化变量”，因此索引键和约束条件的匹配也不仅仅是对 RelOptInfo->baserestrictinfo 的匹配，还需要进行 RelOptInfo->joininfo 的匹配及等价类中保存的约束条件的匹配。

因此，索引键与约束条件的匹配分成了3部分。

- 索引键与 RelOptInfo-> baserestrictinfo（确切地说是 IndexOptInfo->indrestrictinfo）的匹配，匹配结果主要用来生成普通的索引扫描路径，并且为位图扫描生成待选路径。
- 索引键与 RelOptInfo->joininfo 的匹配，匹配结果主要用来生成参数化路径，并且为位图扫描生成待选路径。
- 索引键与等价类中隐含的约束条件的匹配，匹配结果也是用来生成参数化路径，并且为位图扫描生成待选路径。

索引路径的生成是在 create_index_paths 函数中实现的，在这个函数中循环处理了表上的每一个索引，这个循环处理的过程如图6-5所示。

如图 6-6 所示的调用关系可以看出，所有的匹配最终都归结到 match_clause_to_indexcol 函数。



严禁网络传播本PDF，违者责任自负！

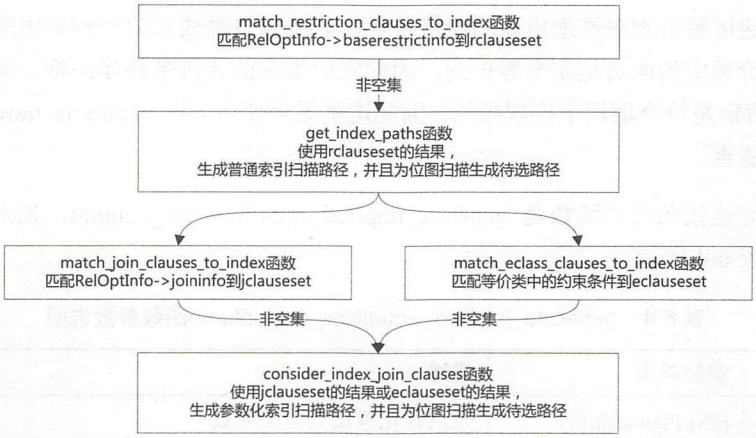


图 6-5　创建索引路径的流程

在索引键和 RelOptInfo-> joininfo 中的子约束条件进行匹配之前，首先要判断 RelOptInfo-> joininfo 中的约束条件是否符合生成参数化路径条件，查询优化器使用 join_clause_is_movable_to 函数进行判断，判断的依据是在假设约束条件一端的操作数为“常量”的情况下，该约束条件是否满足谓词下推的规则，如果约束条件不满足谓词下推规则，则它没有产生参数化路径的可能，也就无须再做它与索引键的匹配，如果该约束条件符合在一端常量化之后满足谓词下推的规则，则可以尝试将索引键与其常量化后的约束条件进行匹配。join_clause_is_movable_to 函数我们在第 4 章和第 6 章都已经做了说明，这里不再重复分析。

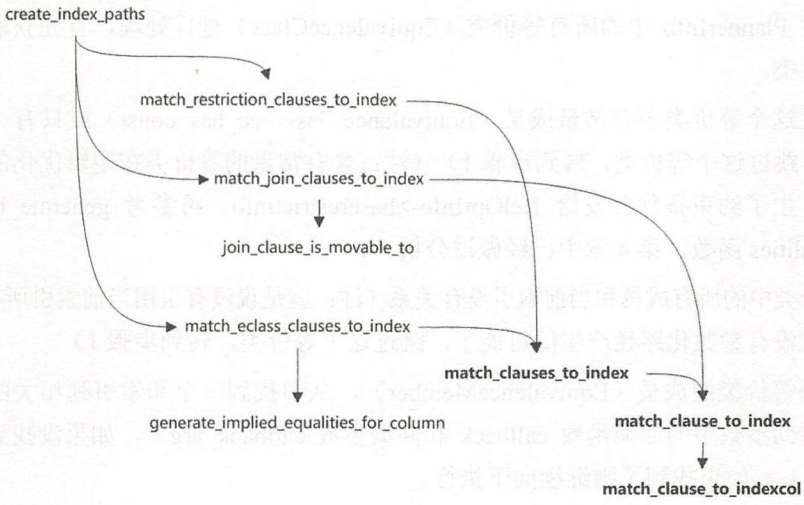


图 6-6　生成索引扫描路径的函数调用关系



严禁网络传播本PDF，违者责任自负！

我们可以通过等价类来推理出新的连接条件，索引键也需要和这些推理出的连接条件进行匹配，因为等价类中的成员是完全等价的，因此它们生成的连接条件如果将一端的操作数“常量化”之后，肯定是符合谓词下推规则的，因此这里无须使用 join_clause_is_movable_to 函数进行谓词下推的检查。

等价类推导连接条件的函数是 generate_implied_equalities_for_column，先来看一下这个函数的参数，如表 6-4 所示。

表 6-4　generate_implied_equalities_for_column函数参数说明

参数名	参数类型	描述
root	[IN] PlannerInfo *	查询优化模块上下文参数
rel	[IN] RelOptInfo *	要匹配的索引所在的表的RelOptInfo结构体
callback	[IN] ec_matches _callback_type *	遍历一个等价类，用等价类的每个成员和callback_arg中提供的索引键匹配
callback_arg	[IN] void *	1. 要匹配的索引的IndexOptInfo 2. 要匹配这个索引的哪个键
prohibited_rels	[IN] Relids	这里的值实际上是rel->lateral_referencers，参数化路径和Lateral都是通过lateral的方法来实现的（最终通过NestloopParam实现），因此要避免互相Lateral的情况

下面来分析一下 generate_implied_equalities_for_column 的实现流程。

1）要对 PlannerInfo 中的所有等价类（EquivalenceClass）进行处理，首先获取其中的一个等价类。

2）如果这个等价类中有常量成员（EquivalenceClass->ec_has_const）或只有一个等价类成员，跳过这个等价类，转到步骤 1）（注：含有常量的等价类在逻辑优化的阶段已经推理产生了约束条件分发给 RelOptInfo->baserestrictinfo，可参考 generate_base_implied_equalities 函数，第 4 章中已经做过分析）。

3）等价类中的所有成员和当前索引没有关系（注：就是说没有引用当前索引所在表的列），也就没有参数化路径产生的可能了，跳过这个等价类，转到步骤 1）。

4）遍历等价类的成员（EquivalenceMember），从中找到一个和索引列相关的等价类成员（借助参数中的回调函数 callback 和回调参数 callback_arg），如果没找到，跳转到步骤 1），如果找到了则继续向下执行。

5）再次遍历等价类成员（EquivalenceMember）。



严禁网络传播本PDF，违者责任自负！

a. 跳过步骤 4）中已经找到的成员，因为自己不能和自己构成连接条件。

b. 跳过仍然与索引相关的等价类成员，避免出现生成的连接条件的两端都是同一个表的列属性的情况。

c. 跳过与 prohibited_rels 中列出的表相关的等价类成员，实际上就是避免两个表出现互为 Lateral 的情况。

d. 跳过自己和自己的父表生成连接条件的可能。

6）用步骤 4）找到的等价类成员和步骤 5）找到的等价类成员生成连接条件。

6.4.2.2　键值与列匹配

match_clause_to_indexcol 函数的参数说明如表 6-5 所示。

表 6-5　match_clause_to_indexcol函数的参数说明

参数名	参数类型	描述
index	[IN] IndexOptInfo*	对应的索引，在匹配的过程中需要记录一些索引的属性
indexcol	[IN] int	要匹配的索引键值
rinfo	[IN] RestrictInfo *	要匹配的子约束条件

在逻辑优化的阶段，我们已经对谓词进行了规范处理，形成的是一棵基于合取范式的表达式树，所以约束条件的最上层一定是通过 AND 串联起来的一个链表（注：也可能只有一个约束条件，这不影响下面的分析结果），链表的每个节点或者是一个单独的表达式（例如 OPExpr），或者是一个以 OR 串联起来的子约束条件链表。例如 A = 1 AND (B = 2 OR C = 3)这样的约束条件，它的构成是两个子约束条件，一个子约束条件是 A=1 形成的一个单独的操作符（OpExpr）表达式，另一个子约束条件是(B = 2 OR C = 3)形成的以 OR 操作符连接的 BoolExpr（boolop:OR_EXPR）表达式。

并不是所有的子约束条件都能进行索引的匹配，match_clause_to_indexcol 函数主要支持 BooleanTest、OpExpr、RowCompareExpr、ScalarArrayOpExpr、NullTest 这几种表达式，因为这些表达式都可以直接或者间接地转换成 indexkey op const 或者 const op indexkey 的形式。

match_clause_to_indexcol 函数对 BooleanTest 类型的表达式做了特殊处理，直接使用 match_boolean_index_clause 函数来进行匹配，match_boolean_index_clause 函数中处理了 3 种情况。

- 直接匹配，表达式本身就是 Var，可以直接匹配成功。
- 带有 NOT 语义的表达式，需要递归处理。
- BooleanTest 表达式，直接使用 BooleanTest 中的 Var 进行匹配。



严禁网络传播本PDF，违者责任自负！

```
bool match_boolean_index_clause(Node *clause, int indexcol,IndexOptInfo *index)
{
   //直接匹配
   if (match_index_to_operand(clause, indexcol, index))
      return true;

   //带有 NOT 语义的 BoolExpr 表达式，匹配其“参数”
   if (not_clause(clause))
      ……
   //对 BooleanTest 表达式进行处理，匹配其“参数”
   else if (clause && IsA(clause, BooleanTest))
      ……
   return false;
}
```

用一个示例来概括上面 3 种情况，示例如下：

```
CREATE TABLE TEST_BOOL(a BOOL, b BOOL, c BOOL, d BOOL);
CREATE INDEX TEST_BOOL_ABC_IDX ON TEST_BOOL(a, b, c);
SELECT * FROM TEST_BOOL WHERE
                a              -- 直接匹配，Var 表达式
                AND NOT b      -- NOT 语义，BoolExpr 表达式
                AND c IS TRUE;  -- BooleanTest 表达式
```

match_clause_to_indexcol 函数对 RowCompareExpr 表达式进行了特殊处理，在 match_rowcompare_to_indexcol 函数中判断索引的匹配性。

- RowCompareExpr 表达式只支持 B 树索引（RowCompareExpr 表达式中比较操作的实现依赖于 B 树索引的操作符族），并且只支持<、<=、>、>=四种操作符（因为=和<>在语法解析的阶段被转换为其他形式，具体可以查阅 make_row_comparison_op 函数）。
- RowCompareExpr 表达式只匹配第一个左操作数和右操作数，这里不匹配其他操作数，例如有约束条件(a, b) > ROW(1, 2)，它会形成一个 RowCompareExpr，那么在匹配时只匹配左操作数 a 和右操作数 1。
- 如果约束条件是 ROW(1, 2) < (a, b)这种形式，则必须有对应的反向操作符（每个操作符在 PG_OPERATOR 系统表中都有一个对应的 OPRCOM 属性），也就是说能够转换成(a, b) > ROW(1, 2)这种形式。

```
static bool match_rowcompare_to_indexcol(……)
{
   //B 树索引
   if (index->relam != BTREE_AM_OID)
```




严禁网络传播本PDF，违者责任自负！

```
        return false;

    //只比较第一个操作数
    leftop = (Node *) linitial(clause->largs);
    ……
    //索引键是左操作数和右操作数的情况
    if (match_index_to_operand(leftop, indexcol, index) &&
        !bms_is_member(index_relid, pull_varnos(rightop)) &&
        !contain_volatile_functions(rightop))
        ……

    //支持>、 >=、 <、 <=四种操作符
    switch (get_op_opfamily_strategy(expr_op, opfamily))
        ……
    return false;
}
```

对于NullTest表达式，需要保证索引本身支持IndexOptInfo->amsearchnulls（与IndexAmRoutine->amsearchnulls 的值相同），然后就可以调用 match_index_to_operand 对匹配性做检查。

OpExpr 表达式和 ScalarArrayOpExpr 表达式进行索引匹配的方式基本相同，因为它们都是比较明显的 indexkey op const 的形式，略有不同的是，ScalarArrayOpExpr 表达式只有 indexkey op const 的形式，没有 const op indexkey 的形式，也就是说从语法上不存在 ANY(ARRAY[1,2,3]) < a 这种形态，因此索引键值一定在 ScalarArrayOpExpr 表达式的左操作数上，match_clause_to_indexcol 函数通过 plain_op 来标识这种情况。另外，ScalarArrayOpExpr 表达式和索引键地匹配只支持 ANY 的形式，即只支持 a > ANY(ARRAY[1,2,3]) 这种形式，不支持 a > ALL(ARRAY[1,2,3])这种形式。

OpExpr 表达式和 ScalarArrayOpExpr 表达式在匹配索引的过程中还需要注意以下两点。

- indexkey 可以是关于一个列属性的简单 Var，也可以是与 indexkey 相关的表达式（不是单纯的 Var），但如果 indexkey 是表达式只会匹配表达式索引，这需要由 match_index_to_operand 函数来检查。
- const 并不一定是真正的常量，只要 const 中不含有易失性函数并且没有引用索引所在表上的属性，就可以宽泛地认为是常量，例如参数化路径，对于 TEST_A.a=TEST_B.b 这样的约束条件，为了能够使用索引扫描，可能会形成 TEST_A.a={Param of TEST_B.b} 这样的约束条件，那么这里也可以认为 TEST_A.a = TEST_B.b 是 indexkey op const 的形式。



严禁网络传播本PDF，违者责任自负！

```
//匹配 indexkey，可以是 Var，或者关于 indexkey 的表达式
//如果是表达式，则只匹配表达式索引
if ( match_index_to_operand(leftop, indexcol, index) &&
   //所谓的 const，只要不引用索引所在的表中的列即可
   !bms_is_member(index_relid, right_relids) &&
   //const 中也不能有易失性函数，例如 RANDOM 函数
   !contain_volatile_functions(rightop))
{
   //is_indexable_operator 判断一下 op 是否适用于索引(PG_AMOP 系统表)
   if (IndexCollMatchesExprColl(idxcollation, expr_coll) &&
      is_indexable_operator(expr_op, opfamily, false))
      return true;

   //对于不符合 is_indexable_operator 的 op，尝试做一些特殊处理
   if (match_special_index_operator(clause, opfamily,
                idxcollation, false))
      return true;
   return false;
}
```

match_special_index_operator 函数处理了一些特殊的情况，主要针对 LIKE、ILIKE、正则表达式几种情况，这里以 text 类型上进行的 LIKE 操作为例，假如约束条件是 a LIKE 'abc'，字符串 'abc' 属于 Pattern_Prefix_Exact 的匹配方式，因此它可以提取出一个 a = 'abc'的约束条件用来匹配索引，示例如下：

```
CREATE TABLE TEST_TEXT(a TEXT, b TEXT, c TEXT, d TEXT);
CREATE INDEX TEST_TEXT_A_IDX ON TEST_TEXT(a);
postgres=# EXPLAIN SELECT * FROM TEST_TEXT WHERE a LIKE 'abc';
                          QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on test_text  (cost=4.17..11.28 rows=3 width=128)
   Filter: (a ~~ 'abc'::text)
   -> Bitmap Index Scan on test_text_a_idx  (cost=0.00..4.17 rows=3 width=0)
        Index Cond: (a = 'abc'::text)
(4 rows)
```

6.4.2.3 生成索引扫描路径

从 RelOptInfo->baserestrictinfo 中匹配出的约束条件会保存到 rclauseset 中，它可以用来支持生成普通索引扫描（IndexScan）、快速索引扫描（IndexOnlyScan），并且还会给位图扫描（BitmapIndexScan）生成待选的路径。

索引扫描路径用 indexPath 结构体来表示，它在 Path“基类”的基础上又增加了一些信息，这些信息主要与索引匹配的约束条件相关。需要注意的是，在 Path 基类中已经保存了 total_cost，而在索引路径结构体上又定义了一个 indextotalcost 变量，这是因为在索引路径的生成过程中，一个索引路径既可能直接被用于索引扫描，还可能会用于位图扫描（位图扫描中也会对索引进行扫描，只不过它是将索引扫描出来的结果保存成为图，避免随机读的问题，BitmapHeapPath 结构体会复用 IndexPath 结构体），这两种扫描的代价计算使用了不同的模型，为了不混淆这两种扫描的代价，使用 IndexPath->indextotalcost 来保存位图扫描的代价，而用 Path->tatal_cost 来保存普通索引扫描和快速索引扫描的代价，IndexPath->indexselectivity 同理。

```
typedef struct IndexPath
{
   Path       path; //“基类”
   IndexOptInfo *indexinfo; //对应的索引的信息
   List       *indexclauses; //能够和索引匹配的约束条件
   List       *indexquals; //能够和索引匹配的约束条件，从 indexclauses 生成出来
   List       *indexqualcols; //每个 indexquals 中的项对应的索引键列表，一一对应
   List       * indexorderbys; //Gist 索引支持的一种情况
   List       *indexorderbycols; //Gist 索引支持的一种情况
   ScanDirection indexscandir; //扫描的方向
   Cost       indextotalcost; //该路径的代价
   Selectivity indexselectivity; //该路径下的选择率
} IndexPath;
```

PostgreSQL 数据库使用 get_index_paths 函数来生成各种索引扫描路径，get_index_paths 函数被 create_index_paths 函数调用，create_index_paths 函数先尝试对索引匹配约束条件，将匹配的约束条件保存在 rclauseset 中，然后使用 rclauseset 作为 get_index_paths 的参数，来生成针对这个索引的所有可能的路径。

我们在匹配约束条件和索引键的时候，对 ScalarArrayOpExpr 表达式进行了匹配，因此 rclauseset 中可能出现 ScalarArrayOpExpr 表达式，在 get_index_paths 函数中，针对 ScalarArrayOpExpr 表达式出现的位置不同也分别做了不同的处理。

skip_nonnative_saop 变量代表的是如果索引不支持对数组进行查找（IndexAmRoutine ->amsearcharray），那么是否就把 ScalarArrayOpExpr 表达式从匹配的约束条件中去掉。如果去掉，就把 skip_nonnative_saop 标记成 true（注：这里还隐含了一个条件，就是如果出现了不支持 IndexAmRoutine->amsearcharray 的索引，而且没有去掉这个 ScalarArrayOpExpr 表达式的情况，那么这时还可以给 BitmapScan 生成位图索引扫描路径）。

skip_lower_saop 变量代表的是当 ScalarArrayOpExpr 表达式不匹配在索引的第一个键值上时，那么是否将 ScalarArrayOpExpr 表达式从匹配的约束条件中去掉，如果不去掉，那么就需要设置 found_lower_saop_clause 变量为 true，这是因为如果 ScalarArrayOpExpr 表达式不匹配在索引的第一个键上，那么应用 ScalarArrayOpExpr 表达式作为索引路径上的一个约束条件后，这个索引扫描路径产生的结果需要是无序的，因为目前查询执行器虽然实现索引扫描时支持 ScalarArrayOpExpr 表达式的求值，但是它的实现方式无法保证这种情况下扫描结果仍然保持有序。

get_index_paths 函数通过 skip_nonnative_saop 变量、skip_lower_saop 变量和 build_index_paths 函数相结合，生成索引扫描路径及位图扫描的待选路径，get_index_path 函数分 3 次调用了 build_index_paths 函数。

- 第 1 次调用：输入了 skip_nonnative_saop 变量、skip_lower_saop 变量，这时如果在 rclauseset 中有 ScalarArrayOpExpr 表达式，如果索引不支持数组查找（IndexAmRoutine ->amsearcharray）或者 ScalarArrayOpExpr 表达式不在索引的第一个键，build_index_paths 生成的索引路径的约束条件中不考虑 ScalarArrayOpExpr 表达式，将 ScalarArrayOpExpr 表达式从约束条件链表中排除出去。
- 第 2 次调用：在 skip_lower_saop == true 的情况下，也就是说第 1 次调用的时候由于 ScalarArrayOpExpr 表达式不是索引的第一个键，导致当时的索引没有考虑 ScalarArrayOpExpr 表达式，这里将 skip_lower_saop 参数设置为 NULL，只输入了 skip_nonnative_saop 变量，也就是说，如果 ScalarArrayOpExpr 表达式不是匹配的索引的第一个键，也考虑为其建立索引扫描路径，但是需要用 found_lower_saop_clause 变量来标识这个索引扫描路径产生的结果是不能保证有序的。
- 第 3 次调用：在 skip_nonnative_saop == true 的情况下，也就是说在索引不支持数组查找（IndexAmRoutine->amsearcharray）的情况下，我们没有将 ScalarArrayOpExpr 表达式考虑进索引路径，这时将 skip_nonnative_saop 参数、skip_lower_saop 参数全部设置为 NULL，仍然可以考虑为其建立索引扫描路径，但是需要注意的是，这里生成的索引扫描路径仅仅留给位图扫描待选。

对于前两次调用产生的扫描路径，可以尝试将其加入普通索引扫描路径列表中，也可以将其保存起来供位图扫描选用（保存到 bitindexpaths 链表），第 3 次调用产生的路径直接保存到 bitindexpaths 链表供位图扫描选择。

```
foreach(lc, indexpaths)
{
```

```
    IndexPath  *ipath = (IndexPath *) lfirst(lc);

    //能够用于普通索引扫描
    if (index->amhasgettuple)
        add_path(rel, (Path *) ipath);

    //保留起来给位图扫描选用
    //（小提示：同一个路径，在普通索引扫描和位图扫描的时候都可能被使用，因此
    // IndexPath 结构体中给位图扫描的代价和选择率保留了单独的变量）
    if (index->amhasgetbitmap &&
        (ipath->path.pathkeys == NIL ||
         ipath->indexselectivity < 1.0))
        *bitindexpaths = lappend(*bitindexpaths, ipath);
}
```

get_index_paths 函数 3 次调用 build_index_paths 函数，build_index_paths 函数则通过输入的参数值的不同，对各种情况进行处理，我们从实现流程和源码分析两个角度来看一下 build_index_paths 函数关于 ScalarArrayOpExpr 表达式部分的处理。

```
//循环处理索引的每一个键
for (indexcol = 0; indexcol < index->ncolumns; indexcol++)
{
    ListCell  *lc;

    //根据索引的键处理 rclauseset 中的每一个约束条件
    foreach(lc, clauses->indexclauses[indexcol])
    {
        RestrictInfo *rinfo = (RestrictInfo *) lfirst(lc);

        //如果约束条件是一个 ScalarArrayOpExpr 表达式
        if (IsA(rinfo->clause, ScalarArrayOpExpr))
        {
            //若果索引不支持 searcharray，也就是索引不能够对数组进行查找
            if (!index->amsearcharray)
            {
                //参数中传进来了 skip_nonnative_saop 变量
                //这时的 skip_nonnative_saop 负责和 get_index_paths 通信
                //它一方面通知 build_index_paths 函数是否跳过这个 ScalarArrayOpExpr 表达式
                //另一方面也反向通知 get_index_paths 函数我是否跳过了 ScalarArrayOpExpr 表达式
                if (skip_nonnative_saop)
                {
                    /* Ignore because not supported by index */
```

```
                *skip_nonnative_saop = true;
                continue;
            }
            //get_index_paths 函数第 3 次调用 build_index_paths 函数才会执行到这里
            Assert(scantype == ST_BITMAPSCAN);
        }

        //ScalarArrayOpExpr 表达式是否出现在索引的第一个键
        if (indexcol > 0)
        {
            //和 skip_nonnative_saop 相似，既是输入参数也是输出参数
            if (skip_lower_saop)
            {
                /* Caller doesn't want to lose index ordering */
                *skip_lower_saop = true;
                continue;
            }

            //当前通过索引进行数组查找，无法保证查找结果有序
            found_lower_saop_clause = true;
        }
    }
    index_clauses = lappend(index_clauses, rinfo);
    clause_columns = lappend_int(clause_columns, indexcol);
    outer_relids = bms_add_members(outer_relids,
                                   rinfo->clause_relids);
  }

  //有些索引要求约束条件必须从索引的第一个键开始
  if (index_clauses == NIL && !index->amoptionalkey)
    return NIL;
}
```

下面看一下这段代码的流程图，如图 6-7 所示。

虽然这里的逻辑看上去有点复杂，但是如果我们跳开 ScalarArrayOpExpr 表达式，build_index_paths 函数实际上也只会进行第一次调用（其他两次都是为第一次调用 build_index_paths 没有使用 ScalarArrayOpExpr 表达式而做的补充），因此读者分析源码的时候不妨把 ScalarArrayOpExpr 表达式排除在外，这样逻辑就会比较清晰了。

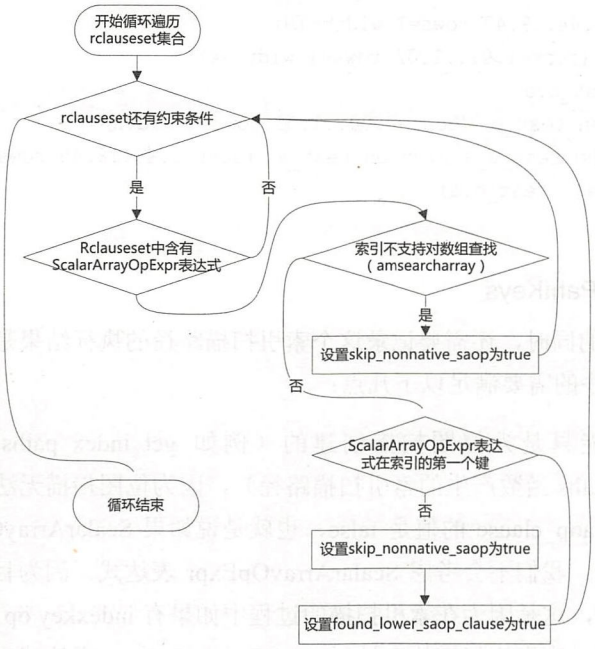


图 6-7　索引扫描路径建立过程

对于参数化的索引扫描路径，参数值每改变一次，这个索引扫描路径就需要执行一次，它执行的次数取决于参数值的“生产者”能够生成多少个参数值。一个路径的执行次数是代价计算时一个非常重要的因素，因此需要对索引扫描路径的执行次数有一个估计值，build_index_paths 函数通过 get_loop_count 函数来获得这个估计值。

在 build_index_paths 函数中，可以获取到可能作为这个索引扫描路径的外表存在的表，例如这个索引所在表的 lateral_relids、索引匹配的条件中涉及的表（参数化路径生成的时候，和索引匹配的约束条件可能是包含内表和外表的连接条件），把它们保存在 outer_relids 中，get_loop_count 函数在 outer_relids 中找到结果集最小的一个，至于为什么选择最小的一个而不选择最大的，大概是基于 PostgreSQL 数据库开发人员的经验。

在 get_loop_count 函数中还对 SemiJoin 的一种情况做了特殊处理，通过 adjust_rowcount_for_semijoins 函数调整了 loop 的次数，从示例的执行计划可以看出，外表的执行次数是分组（Grouping）的数量而不是元组的数量（注意：示例中已经消除了 SemiJoin，转换成了 InnerJoin）：

```
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE a IN (SELECT a FROM TEST_B);
                        QUERY PLAN
-------------------------------------------------------------------------------
```

```
Nested Loop  (cost=1.44..9.47 rows=1 width=19)
  ->  HashAggregate  (cost=1.01..1.02 rows=1 width=4)
        Group Key: test_b.a
        ->  Seq Scan on test_b  (cost=0.00..1.01 rows=1 width=4)
  ->  Index Scan using test_a_a_uidx on test_a  (cost=0.43..8.45 rows=1 width=19)
        Index Cond: (a = test_b.a)
(6 rows)
```

6.4.2.4 索引的 PahtKeys

在创建索引路径的同时，还需要记录这个索引扫描路径的执行结果是否有序，要保证索引扫描产生的结果是有序的需要满足以下几点：

- 这个路径不能只是为位图扫描创建的（例如 get_index_paths 函数中第三次调用 build_index_paths 函数产生的索引扫描路径），因为位图扫描无法保证有序的结果。
- found_lower_saop_clause 的值是 false，也就是说如果 ScalarArrayOpExpr 表达式不在索引的第一个键，我们不会考虑 ScalarArrayOpExpr 表达式。因为目前无法保证结果索引扫描结果有序，这是因为在索引扫描的过程中如果有 indexkey op ScalarArrayOpExpr 表达式这种条件，查询执行模块会针对 ScalarArrayOpExpr 表达式中的每个值进行一次索引扫描，这样产生的结果就无法保证有序了。
- index->sortopfamily != NULL，就是说索引的操作符族保证了索引本身是有序的。

有些情况下，即使索引扫描路径的结果是有序的，也不一定对这个有序的性质进行记录，因为在一些简单的查询中，有序的性质基本用不上，只有在“有用”的情况下索引有序的性质才有记录下来的价值，这里做了两个简单的判断来查看索引有序的性质是否有记录的价值。

- 查询语句中包含连接条件，这就有可能会产生 MergeJoin 这样的连接路径，因为 MergeJoin 需要对连接两端的表进行排序，因此如果索引扫描是有序的，那么就可能省掉 MergeJoin 中的排序，因此记录这个有序的性质是有用的。
- 查询中的Non-SPJ操作提示我们输出结果需要是有序的，例如Non-SPJ操作中有ORDER BY 子句，它会通过 PlannerInfo->query_pathkeys 提示 query_planner 函数生成的查询路径“尽量”满足它的顺序，这样 ORDER BY 操作就可以免掉自己去排序了。

如果索引的扫描结果本身有序，而且查询优化器“认为”这种有序的性质是有用的，那么就会给这个索引扫描路径记录一个 PathKeys。

```
pathkeys_possibly_useful = (scantype != ST_BITMAPSCAN &&//不是为位图扫描生成的路径
                            //不存在 ScalarArrayOpExpr 表达式
                            //ScalarArrayOpExpr 表达式在索引的第一个键
```


非卖品，仅供非商业用途或交流学习使用

```
                        !found_lower_saop_clause &&
                        //查询中有连接操作（这样就可能会有 MergeJoin 路径）
                        //或者“上层 Non-SPJ 操作”对下层有有序的“期望”
                        has_useful_pathkeys(root, rel));
//索引本身有序
index_is_ordered = (index->sortopfamily != NULL);

if (index_is_ordered && pathkeys_possibly_useful)
{
    //创建 PathKeys
    index_pathkeys = build_index_pathkeys(root, index, ForwardScanDirection);
    //从创建的 PathKeys 中选出有用的部分
    useful_pathkeys = truncate_useless_pathkeys(root, rel, index_pathkeys);
    ......
}
```

注意，这里还有一个 index->amcanorderbyop 的判断，目前只有 Gist 索引支持这种特殊情况，这里给出一个示例，有兴趣的读者可以自行分析，这里不展开分析这种情况。

```
CREATE TABLE POINT_TBL(f1 POINT);
CREATE INDEX gpointind ON POINT_TBL USING gist (f1);
postgres=# explain SELECT * FROM point_tbl ORDER BY f1 <-> '0,1';
                        QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using gpointind on point_tbl  (cost=0.13..8.27 rows=7 width=24)
   Order By: (f1 <-> '(0,1)'::point)
(2 rows)
```

创建 PathKeys 是通过 build_index_pathkeys 函数实现的，该函数的流程如图 6-8 所示。

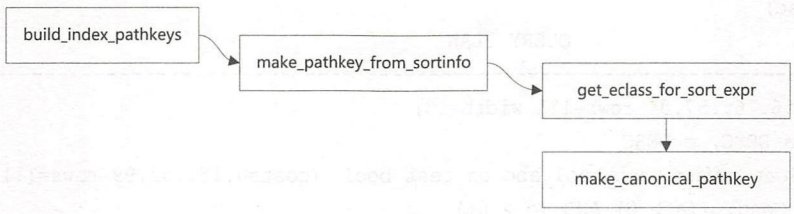


图 6-8　创建索引扫描的 PathKeys

build_index_pathkeys 函数针对索引的每个键去调用 make_pathkey_from_sortinfo 函数，make_pathkey_from_sortinfo 函数则首先调用 get_eclass_for_sort_expr 获得该索引键（列）对应的等价类，然后再使用这个等价类调用 make_canonical_pathkey 函数创建对应的 PathKey。

非卖品，仅供非商业用途或交流学习使用

这里需要明确的是，在 build_index_pathkeys 函数中，对于 bool 类型的索引键，即使没有获得 PathKey，在有些情况下也是可以按照有序考虑的。例如，对于"WHERE indexcol"或"WHERE NOT indexcol"这样的约束条件，索引扫描产生的结果中该列的值是相等的，因此不会影响后续的键值排序，例如：

```
CREATE TABLE TEST_BOOL(a int, b bool, c int, d int);
CREATE INDEX TEST_BOOL_ABC ON TEST_BOOL(a,b,c);
postgres=# explain select * from test_bool where a>0 and b and c>0 order by a desc, c desc;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan Backward using test_bool_abc on test_bool  (cost=0.15..53.55 rows=111 width=13)
   Index Cond: ((a > 0) AND (b = true) AND (c > 0))
   Filter: b
(3 rows)
```

从示例的查询语句可以看出，约束条件 WHERE a > 0 AND b AND c > 0 中的 b 列是 bool 类型，在匹配索引条件的时候能匹配成功，而且在生成索引路径的时候会将其转换成 b=true，保证了索引扫描之后 b 列对应的值全部是相等的，也就是说它满足 match_boolean_index_ clause 函数中提出的要求，所以扫描结果在(a, b, c)三个键值上都是有序的。从执行计划也可以看出 ORDER BY a desc,c desc 这个上层的 Non-SPJ 操作被省略掉了（执行计划中没有 Sort 执行路径），也从侧面证明了索引扫描的 PahtKeys 是(a, b, c)。

但如果约束条件不符合 match_boolean_index_clause 函数的要求，则无法创建基于(a,b,c)三个键的 PathKeys，也就是说无法生成满足“order by a desc, c desc”这样的 PathKey，这时就需要在索引扫描之后继续在(a,c)上做排序操作。

```
postgres=# explain select * from test_bool where a>0 and coalesce(b,true) and c>0 order by
a desc, c desc;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Sort  (cost=56.76..57.04 rows=111 width=13)
   Sort Key: a DESC, c DESC
   ->  Index Scan using test_bool_abc on test_bool  (cost=0.15..52.99 rows=111 width=13)
         Index Cond: ((a > 0) AND (c > 0))
         Filter: COALESCE(b, true)
(5 rows)
```

这里还需要解释一下“规范”的 PathKey 的含义，规范的含义是指生成 PathKey 中的等价类（PathKey->pk_eclass）是“最终”等价类，因为在等价类的建立过程中可能会合并两个等价类，被合并的一方会从等价列的链表（root->eq_classes）中删除，因此被合并的等价类就不是

非卖品，仅供非商业用途或交流学习使用

“最终”等价类，而 PathKey 中的等价类必须是合并之后产生的等价类，因此在 make_canonical_pathkey 函数中，需要对参数中的等价类调整，也就是根据它的 ec_merged 推导出“最终”等价类，也就是规范的等价类。

```
//找到“最终”等价类，也就是“规范”等价类
while (eclass->ec_merged)
   eclass = eclass->ec_merged;

//从已经存在的 PathKeys 中查找，看是否已经存在对应的 PathKey
foreach(lc, root->canon_pathkeys)
{
   pk = (PathKey *) lfirst(lc);
   if (……)
      return pk;
}

//如果不存在，则新建一个 PathKey，注意新建的 PathKey 要放到对应的 MemoryContext 中
oldcontext = MemoryContextSwitchTo(root->planner_cxt);
pk = makeNode(PathKey);
……

//将新建的 PathKey 加入 PathKey 链表
root->canon_pathkeys = lappend(root->canon_pathkeys, pk);
```

build_index_pathkeys 根据索引的键来创建 PathKeys，前面已经介绍过 PahtKeys 对查询优化来说有两个用处，一个是给 MergeJoin 省掉排序操作，另一个是看是不是满足 Non-SPJ 操作的期望（如 ORDER BY 子句），这里需要查看一下这个 PathKeys 是否能满足这两个用处（truncate_useless_pathkeys 函数）。

- 假如两个用处都不满足，这个 PathKeys 没有用，直接放弃就好了。
- 假如满足其中一个用处，还需要查看满足用处的“最短”的 PathKeys，也就是说假如根据索引生成的 PathKeys 在{a, b, c}三个键上，MergeJoin 需要保证在{c}上有序，因此这个 PahtKeys 对 MergeJoin 没有用，而 ORDER BY 操作在{a,b}两个键上，那么最短的 PathKeys 就是{a, b}，这时需要对 PathKeys 做截断。
- 假如两个用处都满足，那么需要在两个“最短”的 Pathkeys 中保留“较长”的一个，假如根据索引生成的 PathKeys 在{a, b, c}三个键上，MergeJoin 需要的 PathKeys 是{a}，而 ORDER BY 操作在{a,b}两个键上，那么最短的 PathKeys 中“较长”的一个是{a, b}，这时也需要将{a, b, c}截断成{a, b}。

非卖品，仅供非商业用途或交流学习使用

6.4.2.5 参数化路径的建立

在处理 rclauseset 结构体之后，还需要处理 jcaluseset 和 eclauseset，rcluaseset 主要用来生成普通的索引扫描路径，而 jclauseset 和 eclauseset 则用来生成普通的参数化索引扫描路径，当然其中的一些路径也会留给位图扫描选用，建立参数化索引扫描路径的主要流程如图 6-9 所示。

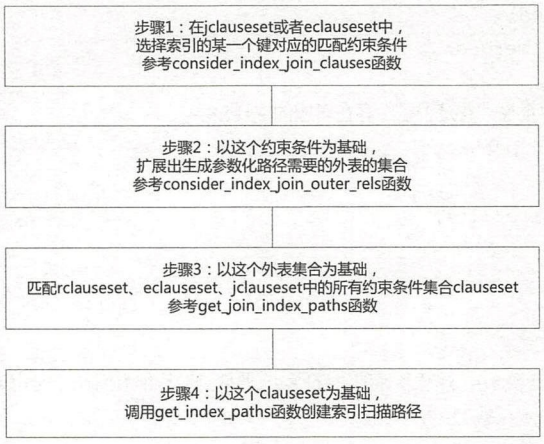


图 6-9 参数化路径的建立过程

通过 consider_index_join_clauses 函数，分别遍历 jclauseset 和 eclauseset 中针对每个索引键的匹配约束条件，然后 consider_index_join_outer_rels 函数针对每个索引键的约束条件“以点带面”，开始探索可能的外表，由于有可能作为外表的组合通常并不多，因此这个探索的复杂度是可控的，PostgreSQL 数据库使用 considered_relids 来保存已经探索过的组合，保证不重复生成同样的组合。

```
//索引的一个键上可能匹配多个约束条件
foreach(lc, indexjoinclauses)
{
   RestrictInfo *rinfo = (RestrictInfo *) lfirst(lc);

   //约束条件中涉及的表除了索引本身的表之外，其他的表都可以考虑当作外表
   Relids       clause_relids = rinfo->clause_relids;
   ListCell   *lc2;

   //considered_relids 记录所有已经处理过的外表集合
   if (bms_equal_any(clause_relids, *considered_relids))
      continue;
```


非卖品，仅供非商业用途或交流学习使用

```
    //这个 foreach 是用当前的约束条件中的表的集合
    //和以前处理过的每个集合组成新的外表集合尝试一下
    foreach(lc2, *considered_relids)
    {
        ……
        //判断：如果是子集关系，那么就没必要组合到一起
        //判断：如果约束条件产生自某个等价类，而且这个等价类已经被使用过了，就跳过
        //……（判断的代码略）

        //为了降低算法的复杂度，对外表集合的长度进行限制
        //也就是说在每处理一个匹配的约束条件，集合数就可以增长 10 个
        //这样如果匹配的约束条件是 2 个，最多可以产生 20 个外表集合
        //（有可能超出，因为下面有不受限制的部分）
        if (list_length(*considered_relids) >= 10 * considered_clauses)
            break;

        //bms_union(clause_relids, oldrelids)：组合成新的外表集合
        get_join_index_paths(bms_union(clause_relids, oldrelids), ……);
    }

    //clause_relids：尝试只用匹配的约束条件生成索引扫描路径（不受长度限制）
    get_join_index_paths(clause_relids,……);
}
```

假如有 3 个表分别是 TEST_A、TEST_B、TEST_C，且 TEST_A 上有索引 TEST_A_AB_IDX(a,b)，针对这样的表有 SQL 语句 SELECT * FROM TEST_A a, TEST_B b, TEST_C c WHERE a.a=b.a and a.b=c.b，那么在匹配 TEST_A_AB_IDX 索引的过程中，a.a=b.a 和 a.b=c.b 两个子约束条件都能被匹配上。

- 子约束条件 a.a=b.a 匹配的是 TEST_A_AB_IDX 的第一个索引键，它涉及的表有{TEST_A，TEST_B}。
- 子约束条件 a.b=c.b 匹配的是 TEST_A_AB_IDX 的第二个索引键，它涉及的表有{TEST_A，TEST_C}。

那么在 consider_index_join_clauses 中需要分两轮来处理这两个子约束条件。

- 第一轮：直接调用 consider_index_join_outer_rels 函数，针对 a.a=b.a 生成参数化路径，并且记录外表集合{TEST_A，TEST_B}（也就是 considered_relids）。
- 第二轮：先使用{TEST_A，TEST_C}和第一轮产生的{TEST_A，TEST_B}组合成{TEST_A，TEST_B，TEST_C}生成参数化路径（循环合并 considered_relids 中的项），

非卖品，仅供非商业用途或交流学习使用

再使用{TEST_A，TEST_C}直接生成参数化路径。

我们可以通过示例看一下其中一个执行计划。

```
create index test_a_ab_idx on test_a(a,b);
postgres=# EXPLAIN SELECT * FROM TEST_A a, TEST_B b, TEST_C c WHERE a.a=b.a and a.b=c.b;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.15..792984.75 rows=158286 width=48)
   ->  Seq Scan on test_c c  (cost=0.00..28.50 rows=1850 width=16)
   ->  Nested Loop  (cost=0.15..410.13 rows=1850 width=32)
         ->  Seq Scan on test_b b  (cost=0.00..28.50 rows=1850 width=16)
         ->  Index Scan using test_a_ab_idx on test_a a  (cost=0.15..0.20 rows=1 width=16)
               Index Cond: ((a = b.a) AND (b = c.b))
(6 rows)
```

参数化路径的具体生成还是调用 get_index_paths 函数来实现的，get_index_paths 前面已经介绍过了，这里不再赘述。

6.4.2.6 创建索引扫描路径

无论是普通索引扫描路径还是参数化的索引扫描路径，都会在 build_index_paths 函数中分 4 次调用 create_index_path 来生成具体的索引扫描路径。

- 生成前向扫描的索引扫描路径。
- 生成前向扫描的并行索引扫描路径。
- 生成后向扫描的索引扫描路径。
- 生成后向扫描的并行索引扫描路径。

我们从 3 个方面对 create_index_path 函数进行说明。

- 记录参数化信息。
- 匹配约束条件修正。
- 索引扫描代价计算。

6.4.2.6.1 记录参数化信息

参数化路径的参数信息使用 ParamPathInfo 结构体保存，它一方面在每个 Path 结构体有一个 ParamPathInfo 类型的成员变量，用于保存这个路径的参数信息；另一方面在参数化路径所对应的表的 RelOptInfo 中也保存了一个以 ParamPathInfo 为节点的 ppilist 链表（List），将 ParamPathInfo 保存到链表的原因是防止重复生成同样的 ParamPathInfo 节点。

非卖品，仅供非商业用途或交流学习使用

参数化信息的生成基于 create_index_path 函数的参数 required_outer，required_outer 是一个表的 rtindex 的集合，它指出了当前路径的参数来自哪个表的。对于扫描路径，参数化信息的生成是通过 get_baserel_parampathinfo 函数来实现的，这个函数会检索 RelOptInfo->ppilist 链表，尝试从中找到已经建立好的参数化信息，如果没有找到，就创建一个新的参数化信息节点 ParamPathInfo。

要建立参数化路径，就需要得到适用于参数化路径的约束条件，在 get_baserel_parampathinfo 函数中，分别从 RelOptInfo->joininfo 或者等价类中获取适合参数化的约束条件。另外，由于参数化路径的约束条件比非参数化路径的约束条件“多”，多出来的约束条件会导致选择率发生变化，因此参数化路径的 rows 不记录在 Path->rows 中，而记录在 Path->param_info-> ppi_rows 中。

join_clause_is_movable_into 函数和 join_clause_is_movable_to 函数虽然名称比较相似，但是在实际使用中有比较大的不同，读者需要注意区分。join_clause_is_movable_to 函数主要用来判断一个约束条件能不能作为索引的“过滤条件”，它主要强调这个约束条件能否“下推”。也就是说，假如将一个约束条件的一端“常量化”，变成 indexkey op const 的形式，这个约束条件能够下推到索引这个“基表”上，只有能下推到索引这个基表上，它才能成为参数化路径。join_clause_is_movable_into 函数则是在生成路径最后的阶段判断哪些条件可以作为当前路径的约束条件，这个时候不关心这个条件能不能用来产生参数化路径，例如：

```
create index test_a_ab_idx on test_a(a,b);
postgres=# explain select * from test_a a, test_b b where a.a=b.a and a.c>b.c;
                              QUERY PLAN
-----------------------------------------------------------------------------------------
 Nested Loop  (cost=0.15..706.13 rows=86 width=32)
   ->  Seq Scan on test_b b  (cost=0.00..28.50 rows=1850 width=16)
   ->  Index Scan using test_a_ab_idx on test_a a  (cost=0.15..0.36 rows=1 width=16)
         Index Cond: (a = b.a)
         Filter: (c > b.c)
(5 rows)
```

示例中的子约束条件 a.c>b.c 肯定是不满足 join_clause_is_movable_to 函数的，因为它和 TEST_A_AB_IDX 索引中的键匹配不上，但是它却满足 join_clause_is_movable_into 函数，因为在生成参数化路径之后，这个约束条件也是可以顺势下推下来的。从执行计划可以看出，子约束条件 a.c>b.c 虽然没有成为索引扫描的匹配条件，但是成为了索引扫描上的一个过滤条件。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

6.4.2.6.2 匹配约束条件的修正

在用索引键和约束条件进行匹配时不仅支持了 OpExpr 表达式，还支持了 BoolExpr、ScalarArrayOpExpr、RowCompareExpr、NullTest 等表达式，这种支持不是无条件的，我们需要在这里将它们转换成“适合”索引扫描机制的约束条件。

例如：

```
CREATE TABLE TEST_BOOL(A BOOL, B BOOL);
CREATE INDEX TEST_BOOL_A_IDX ON TEST_BOOL(A);
postgres=# EXPLAIN SELECT * FROM TEST_BOOL WHERE a;
                              QUERY PLAN
-----------------------------------------------------------------------------------------
 Index Scan using test_bool_a_idx on test_bool  (cost=0.16..67.95 rows=1360 width=2)
   Index Cond: (a = true)
   Filter: a
(3 rows)
```

这里将约束条件中的 a 转换成了 a = true 的形式，实际上就是转换成了索引扫描“喜欢”的 OpExpr，这样它就能清楚地知道自己的操作符族是什么，需要使用什么操作符进行扫描。

我们还可以给出一个 RowCompareExpr 的例子，在匹配 RowCompareExpr 表达式和索引键的时候，只要 RowCompareExpr 的第一个成员满足和索引键匹配，就认为这个 RowCompareExpr 可以和索引匹配上，但是这样的 RowcompareExpr 表达式是无法作为索引扫描约束条件的，需要将匹配的部分提取出来。

```
CREATE TABLE TEST_A(a INT, b INT);
CREATE INDEX TEST_A_A_IDX ON TEST_A(a);
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE (a,b) > row(1,3);
                            QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using test_a_a_idx on test_a  (cost=0.16..61.10 rows=753 width=8)
   Index Cond: (a >= 1)
   Filter: (ROW(a, b) > ROW(1, 3))
(3 rows)
```

对这些“特殊”表达式的处理导致了在 IndexPath 结构体中的 indexclauses 成员变量和 indexquals 成员变量的不同，indexclauses 成员变量和 indexquals 成员变量在没有这些“特殊”表达式的情况下是相同的，在含有特殊表达式的情况下是处理前和处理后的关系。

版权所有，严禁传播，违者自负法律责任！

6.4.2.6.3　索引扫描代价的计算

下面来看一下索引扫描路径代价的计算，从实现上看分成两个部分：

- 一部分是对索引进行扫描的代价，每种类型的索引计算自身的扫描代价的方式都不一样，在 IndexAmRoutine 中定义了 amcostestimate 函数指针，amcostestimate 用来计算每个索引自身的代价，以 B 树索引为例，计算 B 树索引自身扫描代价的函数是 btcostestimate。
- 另一部分是借由索引扫描的结果对堆表进行扫描的代价，这部分代价主要计算了 IO 代价 run_cost 和 CPU 的执行代价 cpu_cost。

在代价计算的过程中，需要估计产生 IO 代价的磁盘页面数，这里需要用到 effective_cache_size 参数，这个参数的含义是：在一个查询中所有表的总页面数中有多少页面是在缓存中的，那么就可以推定当前表有多少页面是在缓存中的（假设分布是平坦的）。

在 index_pages_fetched 函数中获取索引扫描产生 IO 的页面数的方法采用了 Mackert 和 Lohman 的文章 *Index Scans Using a Finite LRU Buffer: A Validated I/O Model* 中建立的模型。

$$PageFetch = \begin{cases} min\left(\dfrac{2TNs}{2T+Ns}, T\right), & T \le b \\ \dfrac{2TNs}{2T+Ns}, & T > b \text{ and } Ns \le 2Tb/(2T-b) \\ b + \left(Ns - \dfrac{2Tb}{2T-b}\right) * \dfrac{T-b}{T}, & T > b \text{ and } Ns > 2Tb/(2T-b) \end{cases}$$

其中：

T 代表的是表中的页面数量。

N 代表的是表中的元组数量。

s 代表约束条件产生的选择率。

b 代表表上的页面已经在缓存中的数量。

假设当前查询中涉及的所有表的总页面数是 total_pages，effective_cache_size 代表的是当前查询中涉及的所有表已经加载到缓存的页面数，那么：

$$b = \frac{effective\_cache\_size}{total\_pages} \times T$$

每种索引类型扫描“自身”的代价（也就是只访问索引的代价，不考虑通过索引扫描堆数

版权所有，严禁传播，违者自负法律责任！

据的代价）在 amcostestimate 指定的函数中计算。下面以 B 树索引的 btcostestimate 函数为例进行分析，btcostestimate 函数的参数情况如表 6-6 所示。

表 6-6　btcostestimate函数的参数说明

参数名	参数类型	描述
root	[IN] PlannerInfo *	对应的索引，在匹配的过程中需要记录一些索引的属性
path	[IN] IndexPath	要匹配的索引键值
loop_count	[IN] double *	索引作为内表，可能要多次扫描
indexStartupCost	[OUT] Cost *	索引自身扫描中的启动代价
indexTotalCost	[OUT] Cost *	索引自身扫描中的整体代价
indexSelectivity	[OUT] Selectivity *	索引产生的选择率
indexCorrelation	[OUT] double *	索引的相关系数，可参考统计信息中的相关系数
indexPages	[OUT] double *	索引的页面数

btcostestimate 函数先对索引扫描匹配的约束条件进行了分析，将分析出的信息保存到 IndexQualInfo 结构体中。

```
typedef struct
{
   RestrictInfo *rinfo;          //被分析的 RestrictInfo
   int          indexcol;        //这个 RestrictInfo 匹配索引的哪个键
   bool       varonleft;         //索引的键在操作符的左端还是右端
   Oid          clause_op;       //操作符的 OID

   //RestrictInfo 中还需要进行运算的对象，例如参数化路径中的参数
   //记录这部分是为了计算索引扫描中的启动代价，如果 RestrictInfo 中
   //包含这种运算对象，那么会在扫描开始前进行运算
   Node       *other_operand;
} IndexQualInfo;
```

通过分析匹配索引的约束条件可以获得索引扫描的 indexBoundQuals，所谓的 indexBoundQuals 就是索引的扫描条件（索引扫描条件已经是按照索引键值排序的了）中的第一个非等值扫描之前的所有的约束条件，这个说法比较抽象。我们看一个例子，对于 TEST_A_ABCDE_IDX（a,b,c,d,e）这样的索引。

- 约束条件 a = 1 AND b = 1 AND c > 1 AND d > 1 对应的 indexBoundQuals 是 a = 1 AND b = 1 AND c > 1，其中 a = 1 AND b = 1 是等值约束条件，c > 1 是第一个非等值的约束条件。

版权所有，严禁传播，违者自负法律责任！

- 约束条件 a > 1 AND b > 1 AND c > 1 AND d > 1 对应的 indexBoundQuals 是 a > 1，因为约束条件中没有等值的约束条件，第一个非等值的约束条件就是 a > 1。
- 约束条件 a = 1 AND c > 1 AND d > 1 对应的 indexBoundQuals 是 a =1，虽然 c > 1 是第一个非等值的约束条件，但中间有一个索引键 b 没有匹配上。

提取 indexBoundQuals 的原因是，我们要访问的索引元组的数量并不是约束条件越多访问的索引元组数量就越少，对于第一个非等值约束条件之后的约束条件，无法对索引元组有好的筛选作用，例如约束条件 a = 1 AND b = 1 AND c > 1 AND d > 1 对应的 indexBoundQuals 是 a = 1 AND b = 1 AND c > 1，如图 6-10 所示。

- a=1 有好的筛选作用是因为它是第一个索引键，符合 a=1 的索引项仍然是聚簇的，也就是在磁盘页面上是连续的。
- b=1 是在 a=1 的基础上的等值操作，因为通过 a=1 的过滤之后，b 列的数据仍然还是有序的，因此 b=1 筛选出的索引项也是聚簇的。
- c>1 是在 a=1 AND b=1 的基础上的，因为 a=1 AND b=1 是聚簇的，所以在 a=1 AND b=1 基础上的数据中，索引键 c 的值目前仍然是有序的，因此 c > 1 产生的结果也是聚簇的。
- a = 1 AND b = 1 AND c > 1 筛选产生的结果是聚簇的，但是在这个结果的基础上，索引键 d 对应的值不再有序了，因此约束条件 a = 1 AND b = 1 AND c > 1 AND d > 1 在使用约束条件进行扫描的时候，并不能精确地定位到 d > 1 的值，而是只能在 a = 1 AND b = 1 AND c > 1 的基础上对 d > 1 做筛选，因此要访问的索引项的数量需要以 indexBoundQuals，也就是 a = 1 AND b = 1 AND c > 1 访问的索引项数量为准。

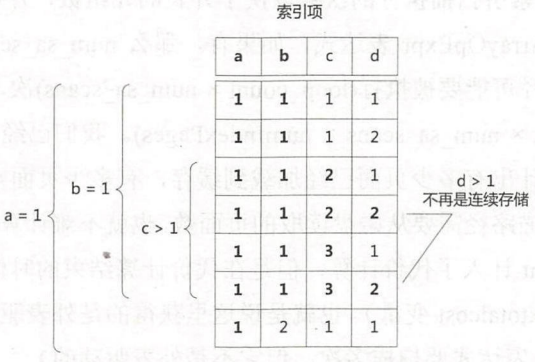


图 6-10　索引条件的边界

在获得 indexBoundQuals 约束条件的过程中，还会记录 eqQualHere、found_saop、found_is_null_op 变量，通过这些变量来确定索引访问的一个特殊情况，就是索引扫描路径如果使用的是

版权所有，严禁传播，违者自负法律责任！

唯一索引且约束条件中包含所有索引键且约束条件是等值约束条件，那么扫描路径执行后产生的结果只有一条元组，除了这种特殊情况之外，就需要使用 indexBoundQuals 来获得一次索引扫描需要访问的索引项的数量（numIndexTuples 变量）。

然后 btcostestimate 函数调用 genericcostestimate 函数获得索引扫描所需要的参数。

- num_sa_scans：索引约束条件中 ScalarArrayOpExpr 表达式的长度，因为 PostgreSQL 数据库目前对约束条件中的 ScalarArrayOpExpr 表达式的执行方式是 ScalarArrayOpExpr 数组中的每一项都要扫描一次索引，因此需要计算 num_sa_scans 来确定索引扫描的次数。
- indexSelectivity：索引上匹配的约束条件（indexQuals）的选择率，注意区分 indexBoundQuals 的选择率，indexBoundQuals 的选择率只用来计算 numIndexTuples。
- numIndexPages：使用 numIndexTuples 估计出来的索引访问页面数。

$$numIndexPages = \frac{numIndexTuples}{index \rightarrow tuples} \times \text{index} \rightarrow \text{pages};$$

- indexStartupCost：启动代价，计算 IndexQualInfo->other_operand 中的表达式代价作为启动代价。
- indexTotalCost：一次索引扫描的整体代价。

下面介绍 indexTotalCost 的计算方法，它又分为 IO 代价和 CPU 代价。

IO 代价：对于一个索引扫描路径，这里有一个需要访问多少次的问题，我们已经计算过 loop_count，也就是说索引扫描执行的次数取决于外表的元组数，另外还取决于索引匹配的约束条件中有没有 ScalarArrayOpExpr 表达式。如果有，那么 num_sa_scans 也是索引扫描的次数，因此一个索引扫描路径可能要被执行(loop_count × num_sa_scans)次，进而可以得到需要扫描的总页面数 (loop_count × num_sa_scans × numIndexPages)。我们已经介绍过 index_pages_fetched 函数，这个函数会估计出有多少页面已经加载到缓存，有多少页面需要从磁盘读取，这样我们就能知道这个索引扫描路径需要从磁盘读取的页面数，也就不难计算 IO 的代价。需要注意的是，这里虽然将 loop_count 计入了代价计算，但是在代价计算结束的时候又将计算出来的代价除以 loop_count（注意 indextotalcost 变量），也就是说这里获得的是外表驱动一次索引扫描的代价（虽然 ScalarArrayOpExpr 表达式要扫描多次，但它不是外表驱动的）。

对于 CPU 代价，因为我们已经知道了 numIndexTuples，而且也知道了 num_sa_scans，也就知道了要访问多少个索引项（numIndexTuples × num_sa_scans），每个索引项上的代价有索引项的基准代价（cpu_index_tuple_cost）和在索引项上应用匹配的约束条件带来的“表达式代价

版权所有，严禁传播，违者自负法律责任！

（QualCost）”，因此 CPU 代价就是（numIndexTuples × num_sa_scans ×（cpu_index_tuple_cost + 表达式代价））。

在调用 genericcostestimate 函数后，btcostestimate 函数还需要计算索引的相关系数，这里重点关注的是索引第一个键对应的统计信息的相关系数，如果索引只有一个键，那么相关系数就是统计信息中的相关系数，如果索引有多个键，那么索引的相关系数就是第一个键的相关系数 ×0.75。

索引扫描的第一部分代价通过 btcostestimate 函数已经计算完成了，第二部分代价计算主要是在 cost_index 函数中完成的，它主要还是计算 IO 代价和 CPU 代价。

下面先来看一下堆表扫描的 IO 代价，它的计算思想是计算一个最大的 IO 代价和一个最小的 IO 代价：最大的 IO 代价是指通过索引扫描堆表的时候完全是随机的，也就是说它们是完全不相关的（相关系数是 0），这时的代价是“大”的；最小的 IO 代价是指索引扫描的是和堆表的数据完全正相关的，也就是说堆表的访问不是随机的，这时的代价是“小”的。而我们已经在 btcostestimate 函数中获得了相关系数 ρ（ρ 是相关系数，ρ^2 叫判定系数，PostgreSQL 目前计算代价的时候用的是 ρ^2），那么我们就可以借由这些推算出 IO 代价。

```
io_cost = max_IO_cost + ρ² * (min_IO_cost - max_IO_cost);
```

我们来看一下 max_IO_cost 和 min_IO_cost 的计算方法。

```
if (loop_count > 1)
{
   //使用 index_pages_fetched 估计会产生 IO 的页面数
   pages_fetched = index_pages_fetched(tuples_fetched * loop_count,
                                       baserel->pages,
                                       (double) index->pages,
                                       root);

   //allvisfrac 是专为快速索引扫描设计的，它就是为了估计索引扫描要访问哪些页面用的
   if (indexonly)
      pages_fetched = ceil(pages_fetched * (1.0 - baserel->allvisfrac));

   rand_heap_pages = pages_fetched;

   //这里使用 spc_random_page_cost，因为最大 IO 是“完全不相关”的，全部是随机 IO
   max_IO_cost = (pages_fetched * spc_random_page_cost) / loop_count;

   //计算最小 IO 要访问的总页面数
```



版权所有，严禁传播，违者自负法律责任！


```
    pages_fetched = ceil(indexSelectivity * (double) baserel->pages);

    //获取总页面数中有多少需要从磁盘读取
    //注意这里输入的参数是pages_fetched×loop_count，和上面的tuples_fetch×loop_count不同
    //这是因为在完全相关的情况下，扫描的页面数就是表的page数量×索引的选择率，这里调用
    //index_pages_fetched函数再处理一下是为了体现哪些页面是在缓存里的
    pages_fetched = index_pages_fetched(pages_fetched * loop_count,
                                        baserel->pages,
                                        (double) index->pages,
                                        root);

    if (indexonly)
        pages_fetched = ceil(pages_fetched * (1.0 - baserel->allvisfrac));

    //注意这里仍然使用的是随机IO访问的代价基准单位
    //PostgreSQL认为index_pages_fetched已经考虑了缓存的效果
    min_IO_cost = (pages_fetched * spc_random_page_cost) / loop_count;
}
else
{
    //注意：在loop_count == 1的时候，最小IO的计算使用的是顺序访问的代价基准单位
    min_IO_cost += (pages_fetched - 1) * spc_seq_page_cost;
}
```

cost_index 中的 CPU 代价主要是堆表上的元组的处理代价（cpu_tuple_cost）和应用在元组上的约束条件的表达式代价，因为约束条件中和索引匹配的约束条件已经在 btcostestimate 计算过代价，因此这里需要将这部分约束条件剥离出去，cost_index 通过 extract_nonindex_conditions 来获得未匹配到索引上的约束条件。例如下面示例中的 SQL 语句，约束条件 a.c=b.c 没有应用到索引扫描上，它是在索引扫描之后进行过滤的约束条件。

```
create index test_a_ab_idx on test_a(a,b);
postgres=# explain select * from test_a a, test_b b where a.a=b.a and a.c=b.c;
                              QUERY PLAN
-----------------------------------------------------------------------------------------
 Nested Loop  (cost=0.15..706.13 rows=86 width=32)
   ->  Seq Scan on test_b b  (cost=0.00..28.50 rows=1850 width=16)
   ->  Index Scan using test_a_ab_idx on test_a a  (cost=0.15..0.36 rows=1 width=16)
         Index Cond: (a = b.a)
         Filter: (b.c = c)
(5 rows)
```




版权所有，严禁传播，违者自负法律责任！

这里还需要考虑并行索引扫描的代价分配问题，分配的方法和 cost_seqscan 函数相似，这里不再赘述。

6.4.3　位图扫描

位图扫描是针对索引扫描的一个优化，它通过建立位图的方式将原来的随机堆表访问转换成了顺序堆表访问，索引必须支持建立位图（IndexAmRoutine->amgetbitmap）才能用于建立位图扫描路径。

在索引扫描路径创建的过程中，一直同时在为位图扫描路径生成待选路径（保存在 bitindexpaths 中），但是只有这些待选路径还不够，因为索引和约束条件匹配的要求比较严格，只支持 OpExpr、ScalarArrayOpExpr、NullTest、BooleanTest、RowCompareExpr 等几种简单的表达式，除了这些表达式，位图扫描还能支持一些更为复杂的情况。

例如约束条件中的 OR 相关的子约束条件，对于 A > 1 OR A > 2 这样的约束语句，即使 A 是索引的键，在创建索引扫描路径时也无法和索引匹配成功，但是位图扫描不同，位图扫描虽然也借助了索引，但它通过索引来生成位图，而位图本身具有交集和并集的操作，因此对于 A > 1 OR A > 2 这样的约束条件可以通过位图的并集来实现，例如：

```
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE A > 1 OR A > 2;
                          QUERY PLAN
-----------------------------------------------------------------------------------
 Bitmap Heap Scan on test_a  (cost=18.07..46.57 rows=1028 width=16)
   Recheck Cond: ((a > 1) OR (a > 2))
   -> BitmapOr  (cost=18.07..18.07 rows=1233 width=0)
        -> Bitmap Index Scan on test_a_a_idx  (cost=0.00..8.78 rows=617 width=0)
             Index Cond: (a > 1)
        -> Bitmap Index Scan on test_a_a_idx  (cost=0.00..8.78 rows=617 width=0)
             Index Cond: (a > 2)
(7 rows)
```

从示例中的执行计划可以看出，位图扫描先借助索引 TEST_A_A_IDX 和子约束条件 A > 1 生成了一个位图，再借助 TEST_A_A_IDX 和子约束条件 A > 2 生成了另一个位图，然后对这两个位图进行了并集操作 BitmapOr，使用 BitmapOr 的结果对 TEST_A 表进行了基于位图的堆（顺序）扫描，它的路径组成如图 6-11 所示。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

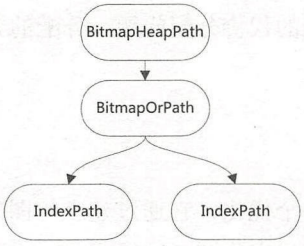


图 6-11　BitmapOr 路径

而对于 AND 操作，位图扫描也可以尝试做交集操作 BitmapAnd，例如对于约束条件 A > 1 AND B > 1，假设属性 A 和属性 B 分别有对应的索引 TEST_A_A_IDX 和 TEST_A_B_IDX，那么就有可能生成执行计划（注：作者通过修改代价计算方式生成了示例中的路径，读者可以自行尝试生成路径，但由于代价的筛选作用，不一定生成同样的路径）。

```
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE B >100 AND A > 200;
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Bitmap Heap Scan on test_a  (cost=17.91..31.00 rows=206 width=16)
   Recheck Cond: ((b > 100) AND (a > 200))
   -> BitmapAnd  (cost=17.91..17.91 rows=206 width=0)
        -> Bitmap Index Scan on test_a_b_idx  (cost=0.00..8.78 rows=617 width=0)
             Index Cond: (b > 100)
        -> Bitmap Index Scan on test_a_a_idx  (cost=0.00..8.78 rows=617 width=0)
             Index Cond: (a > 200)
(7 rows)
```

示例中的路径的组成如图 6-12 所示。

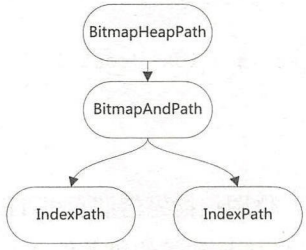


图 6-12　BitmapAnd 路径

由两个示例的路径组成可以看出，BitmapAnd 和 BitmapOr 操作可以将原来“扁平的”“一次扫描”的索引扫描路径扩展成“树状的”“多次扫描”的位图扫描路径，这棵树的叶子节点一定是 IndexPath，只不过这个 IndexPath 负责生成的是位图，而不再去进行堆表扫描，堆表扫

严禁网络传播本PDF，违者责任自负！

描的工作留给 BitmapHeapPath 来做。

位图扫描的顶层一定是 BitmapHeap 扫描路径，它是把 BitmapOr 操作和 BitmapAnd 操作的位图结果结合起来，再次对堆表进行扫描的过程。另外，BitmapOr 操作和 BitmapAnd 操作也是以路径的方式存在的，PostgreSQL 数据库为其分别规定了路径结构体。

```
//位图扫描最顶层的一定是位图堆（顺序）扫描
typedef struct BitmapHeapPath
{
   Path       path;
   Path      *bitmapqual;   /* IndexPath, BitmapAndPath, BitmapOrPath */
} BitmapHeapPath;

//BitmapAnd 操作的扫描路径
typedef struct BitmapAndPath
{
   Path       path;
   List      *bitmapquals;    /* IndexPaths and BitmapOrPaths */
   Selectivity bitmapselectivity;
} BitmapAndPath;

//BitmapOr 的扫描路径
typedef struct BitmapOrPath
{
   Path       path;
   List      *bitmapquals;    /* IndexPaths and BitmapAndPaths */
   Selectivity bitmapselectivity;
} BitmapOrPath;
```

对于 A > 1 AND (A > 2 OR A > 3)这样的约束条件，如果属性 A 是索引上的第一个键，在没有位图扫描路径的情况下，索引扫描只能匹配 A > 1 这个子约束条件，这是因为索引扫描匹配约束条件时将(A > 2 OR A > 3)视为一个整体，不能对其进行拆分，而(A > 2 OR A > 3)也不是简单的 indexkey op const 的形式，也就无法成功地匹配索引，因此它是作为一个过滤条件存在的。但是我们知道位图扫描是能支持(A > 2 OR A > 3)这样形式的约束条件的，那么就可以尝试把这种情况找出来，用来生成 BitmapOr 方式的位图扫描路径。也就是说，我们现在要把索引扫描路径生成过程中没有匹配上的 OR 操作类型的约束条件挑出来，给它们生成位图扫描路径，然后也把它们加入 bitindexpaths 中（bitindexpaths 最终用于生成 BitmapHeapPath）。

6.4.3.1　BitmapHeapPath 的生成流程

这个挑选 OR 操作的过程仍然分成了生成非参数化路径（保存到 bitindexpaths 中待选）和

严禁网络传播本PDF，违者责任自负！

参数化路径（保存到 bitjoinpaths 中待选）两部分，我们先查看非参数化路径的部分。

```
//RelOptInfo->baserestrictinfo 中有没有匹配到索引上的子约束条件，把它们找出来
//生成给位图用的 BitmapOrPath，保存到 bitindexpahts 中
indexpaths = generate_bitmap_or_paths(root, rel, rel->baserestrictinfo, NIL);
bitindexpaths = list_concat(bitindexpaths, indexpaths);

......

if (bitindexpaths != NIL)
{
   Path      *bitmapqual;
   BitmapHeapPath *bpath;

   //在逻辑优化阶段对谓词进行了处理，能够保证约束条件的顶层是 AND 操作
   //choose_bitmap_and 的作用是从 bitindexpaths 中选择“合适”的路径组成 BitmapAndPath
   bitmapqual = choose_bitmap_and(root, rel, bitindexpaths);

   //建立 BitmapHeapPath
   bpath = create_bitmap_heap_path(root, rel, bitmapqual, rel->lateral_relids, 1.0, 0);
   add_path(rel, (Path *) bpath);

   //考虑并行查询
   if (rel->consider_parallel && rel->lateral_relids == NULL)
      create_partial_bitmap_paths(root, rel, bitmapqual);
}
```

对于参数化路径的部分，我们在生成索引扫描路径的时候，在 match_join_clauses_to_index 函数中已经对可以作为参数化位图扫描路径的 OR 操作进行了收集，把它们保存在了 joinorclauses 中，因此可以直接使用 joinorclauses 来生成 bitjoinpaths 集合，bitjoinpaths 链表中保存的都是能够作为参数化路径的扫描路径。

```
//joinorclauses 是我们在 match_join_clauses_to_index 中“顺便”生成的
//生成参数化的 BitmapOrPath，保存到 bitjoinpaths 中
//注意这里还输入了 RelOptInfo->baserestrictinfo 作为参数
indexpaths = generate_bitmap_or_paths(root, rel,joinorclauses, rel->baserestrictinfo);
bitjoinpaths = list_concat(bitjoinpaths, indexpaths);
```

需要注意的是，在使用 joinorclauses 作为输入参数的同时，还输入了 RelOptInfo->baserestrictinfo 作为 other_clauses 参数，我们通过一个例子来看一下它的含义，假如在 TEST_A 表上有索引 TEST_A_A_IDX 和 TEST_A_B_IDX。

严禁网络传播本PDF，违者责任自负！

```
CREATE INDEX TEST_A_A_IDX ON TEST_A(a);
CREATE INDEX TEST_A_B_IDX ON TEST_A(b);
```

对于 SQL 语句 SELECT * FROM TEST_A a, TEST_B b WHERE a.a > 1 and (a.a = b.a or a.b = 1)按照我们的预想，它可能是这样的执行计划。

```
Nestloop Join
          ->SeqScan (TEST_B)
          ->BitmapHeapScan
                ->BitmapAnd
                      ->BitmapIndexScan(TEST_A_A_IDX)(a > 1)
                      ->BitmapOr
                            ->BitmapIndexScan(TEST_A_A_IDX) (a.a = Param of b.a)
                            ->BitmapIndexScan(TEST_A_B_IDX) (a.b = 1)
```

而实际上我们还可以生成下面这种执行计划。

```
postgres=# EXPLAIN SELECT * FROM TEST_A a, TEST_B b WHERE a.a > 1 and ( a.a = b.a or a.b =
1);
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Nested Loop  (cost=10000000004.56..10000016364.76 rows=11386 width=32)
   ->  Seq Scan on test_b b  (cost=10000000000.00..10000000028.50 rows=1850 width=16)
   ->  Bitmap Heap Scan on test_a a  (cost=4.56..8.77 rows=6 width=16)
        Recheck Cond: (((a = b.a) AND (a > 1)) OR (b = 1))
        Filter: (a > 1)
        ->  BitmapOr  (cost=4.56..4.56 rows=12 width=0)
             ->  Bitmap Index Scan on test_a_a_idx  (cost=0.00..0.18 rows=3 width=0)
                  Index Cond: ((a = b.a) AND (a > 1))
             ->  Bitmap Index Scan on test_a_b_idx  (cost=0.00..4.22 rows=9 width=0)
                  Index Cond: (b = 1)
(10 rows)
```

从执行计划可以看出，子约束条件 a > 1 和 a.a = b.a 组合到了同一个 BitmapIndexScan 中，a > 1 就是通过 other_clauses 传递进来的。

bitjoinpaths 中保存了参数化路径集合，我们看一下 PostgreSQL 如何使用这些路径来生成位图扫描路径。

```
if (bitjoinpaths != NIL)
{
   ......
   //path_outer 记录的是 bitjoinpaths 中每个路径的 required_outer
```


严禁网络传播本PDF，违者责任自负！

```
    //all_path_outers 记录的是 path_outer 去重之后的结果
    path_outer = all_path_outers = NIL;
    foreach(lc, bitjoinpaths)
    {
        Path      *path = (Path *) lfirst(lc);
        Relids       required_outer;

        required_outer = get_bitmap_tree_required_outer(path);
        path_outer = lappend(path_outer, required_outer);
        if (!bms_equal_any(required_outer, all_path_outers))
            all_path_outers = lappend(all_path_outers, required_outer);
    }

    //all_path_outers 是去重之后的结果，也就是说我们为每个独立的 required_outer
    //生成一个参数化位图扫描路径
    foreach(lc, all_path_outers)
    {
        ......

        //把 required_outer 相同的参数化路径都提取出来
        this_path_set = NIL;
        forboth(lcp, bitjoinpaths, lco, path_outer)
            ......

        //使用“required_outer 相同的参数化路径”
        //和 bitindexpaths 作为生成位图扫描路径的待选路径
        this_path_set = list_concat(this_path_set, bitindexpaths);

        //建立 BitmapAndPath
        bitmapqual = choose_bitmap_and(root, rel, this_path_set);

        required_outer = get_bitmap_tree_required_outer(bitmapqual);
        loop_count = get_loop_count(root, rel->relid, required_outer);
        //建立 BitmapHeapPath
        bpath = create_bitmap_heap_path(root, rel, bitmapqual,
                                        required_outer, loop_count, 0);
        add_path(rel, (Path *) bpath);
    }
}
```


严禁网络传播本PDF，违者责任自负！

6.4.3.2　BitmapOrPath 的生成流程

对于约束条件中无法和索引匹配的 OR 子句可以通过 generate_bitmap_or_paths 函数为其生成 BitmapOrPath 路径。

```
static List *
generate_bitmap_or_paths(PlannerInfo *root, RelOptInfo *rel,
                         List *clauses, List *other_clauses)
{
   ......
   //主要为参数 clauses 中的 OR 子句生成 BitmapOrPath
   //但在过程中如果能用得上 other_clauses 中的约束条件，也可以使用
   all_clauses = list_concat(list_copy(clauses), other_clauses);

   //clauses 中可能有多个 OR 子句
   foreach(lc, clauses)
   {
      ......
      //如果不是 OR 子句而且能匹配上索引，那么肯定建立索引扫描路径了，
      //这种索引扫描路径肯定已经加入 bitindexpaths 中待选了
      if (!restriction_is_or_clause(rinfo))
         continue;

      //遍历 OR 子句中的子约束条件，OR 子句的子约束条件可能是
      //“独立的约束条件（不带有 AND 或 OR 的子约束条件）”，
      //也可能是 AND 子句，因此在处理 OR 子句的每个子约束条件之后，
      //都调用 choose_bitmap_and 函数，为其生成一个 BitmapAndPath
      //（注：如果是独立子约束条件，实际上在 choose_bitmap_and 函数中会马上原值返回，
      //不会生成 BitmapAndPath 路径）
      pathlist = NIL;
      foreach(j, ((BoolExpr *) rinfo->orclause)->args)
      {
         ......
         //处理 AND 子句
         if (and_clause(orarg))
         {
            List      *andargs = ((BoolExpr *) orarg)->args;
            indlist = build_paths_for_OR(root, rel, andargs, all_clauses);……
         }
         else//处理“独立的子约束条件”
         {  ……
            indlist = build_paths_for_OR(root, rel, orargs, all_clauses);
```


严禁网络传播本PDF，违者责任自负！

```
            }

            //如果 OR 子句中有一个子约束条件无法生成待选路径，那么整个 OR 子句就是不可用的
            if (indlist == NIL)
            {
                pathlist = NIL;
                break;
            }

            //OR 子句的每个子约束条件都按照 AND 子句处理（如果不是 AND 子句会原值返回，
            //不生成 BitmapAndPath）
            bitmapqual = choose_bitmap_and(root, rel, indlist);
            pathlist = lappend(pathlist, bitmapqual);
        }

        //OR 子句的所有子约束条件生成的待选路径都在 pathlist 中，使用 pathlist 建立这个
        //OR 子句对应的 BitmapOrPath
        if (pathlist != NIL)
        {
            bitmapqual = (Path *) create_bitmap_or_path(root, rel, pathlist);

        //参数 clauses 中可能有多个 OR 子句，每个 OR 子句对应的 BitmapOrPath 都保存在 result 中
            result = lappend(result, bitmapqual);
        }
    }

    return result;
}
```

另外，build_paths_for_OR 函数的逻辑就是对 OR 类型的约束条件匹配索引建立“待选”路径的过程，这个过程和索引扫描路径的建立非常相似，读者可以自行分析。

6.4.3.3 BitmapAndPath 的生成流程

在 OR 类型的约束条件中，如果出现了 AND 子句，那么就会调用 choose_bitmap_and 函数来生成 BitmapAndPath。另外，由于查询语句的约束条件经过规范化处理后是合取范式的形式，因此它的执行计划的顶层通常是 BitmapAndPath 的形式（请忽略一些特例，比如 A OR B 这种形式的约束条件产生的扫描路径的顶层就不是合取范式），它使用 choose_bitmap_and 函数来生成最终的 BitmapAndPath。

由于在生成索引扫描路径的时候生成了多个待选路径，在处理 OR 子句的过程中又在

严禁网络传播本PDF，违者责任自负！

generate_bitmap_or_paths 函数中生成了多个 BitmapOrPath 作为待选路径，那么如此多的待选路径我们该如何用来组合一个 BitmapAndPath 呢？PostgreSQL 数据库采用了 3 个阶段。

- 筛选阶段：如果两个待选路径“效果”相同，那么进行优胜劣汰。
- 排序阶段：将“效果”不同的优胜路径排序。
- 组合阶段：按照排序结果进行组合，在组合的过程中如果发现在增加一个路径之后代价反而升高，那么就淘汰这个路径。

6.4.3.3.1　筛选阶段

所谓的“效果”相同是指两个路径具有同样的约束条件集合，这是由于之前已经大量生成了“效果”相同的路径，例如在 TEST_A 表上有索引 TEST_A_A_IDX(a) 和索引 TEST_A_AB_IDX(a, b)，那么对于 a op const 这样的约束条件，两个索引是都能匹配上的，因此会生成两个待选路径，但是这两个待选路径的约束条件是一样的，因此在 choose_bitmap_ands 函数里会对其进行筛选，筛选的方式是估算两个路径的代价，选择代价比较低的一个保留，代价比较高的就放弃。

那么如何判断它们具有相同的“效果”呢？就是把它们的约束条件位图化（这个位图和位图扫描的位图没有关系，只是借助 Bitmapset 结构体来实现的一种比较方式），通过分析所有待选路径的所有约束条件，为每个约束条件进行“编号”，这个编号就是这个约束条件的唯一标识，每个编号在位图中占一位，如果两个路径的位图相同，那么就代表它们具有相同编号的约束条件，也就是说它们具有相同的约束条件，这个过程是在 classify_index_clause_usage 函数中实现的，它创建了一个链表，每当遇到一个约束条件，就在链表中查找这个约束条件，如果能找到，那么这个约束条件的编号就是它在链表中的位置下标，如果找不到这个约束条件，那么就把它加入链表的尾部，链表尾部的下标就是这个约束条件的编号。

6.4.3.3.2　排序阶段

在筛选阶段之后就可以对筛选出来的路径进行排序了，排序使用的是快速排序的方法，排序的比较函数是 path_usage_comparator，它的排序规则是代价低的路径优先，如果代价相同，选择率低的优先。

6.4.3.3.3　组合阶段

在排序之后，就可以对已经存在的路径进行组合了，组合的方法是一个具有 $O(N^2)$时间复杂度的算法，如果有 A、B、C、D 共 4 条路径，那么它的组合顺序是这样的。

- A 与 B、C、D。

非卖品，仅供非商业用途或交流学习使用

- B 与 C、D。
- C 与 D。

它的组合逻辑是这样的，A 和 B 组合，如果组合之后的代价低于 A 原来的代价，那么就保留 A、B 组合的结果，如果组合之后的代价反而升高了，那么就将 B 淘汰，然后按照同样的逻辑选择是否与 C 组合。

```
//newcost 是组合后的估计代价
//costsofar 是组合前的代价
if (newcost < costsofar)
{
   //如果组合后的代价更低，则保留组合的结果，同时更新代价信息
   costsofar = newcost;
   qualsofar = list_concat(qualsofar, list_copy(pathinfo->quals));
   qualsofar = list_concat(qualsofar, list_copy(pathinfo->preds));
   clauseidsofar = bms_add_members(clauseidsofar,
                           pathinfo->clauseids);
   lastcell = lnext(lastcell);
}
else
{
   //否则淘汰新加入的路径
   paths = list_delete_cell(paths, lnext(lastcell), lastcell);
}
```

6.4.3.4 位图扫描代价计算

位图扫描实际上有 4 种不同的路径，分别是 BitmapHeapPath、BitmapOrPath、BitmapAndPath、IndexPath。

位图扫描中的 IndexPath，它一定是位图扫描路径的“树”中的叶子节点，它和普通的索引扫描路径不同，普通的索引扫描路径的代价记录在((Path*)IndexPath)->total_cost 中，而位图扫描中的 IndexPath 的代价记录在 IndexPath->indextotalcost 中，在创建索引扫描路径计算代价时，已经通过 cost_index 函数计算了代价，通过分析 cost_index 函数的代码可以看出，IndexPath->indextotalcost 中记录的代价是 btcostestimate 函数返回的代价，也就是扫描索引自身的代价，这部分代价不包括对堆表扫描的代价。

位图扫描中 BitmapOrPath 的代价计算是在 cost_bitmap_or_node 函数中完成的，BitmapAndPath 的代价计算是在 cost_bitmap_and_node 函数中完成的，它们都需要计算两部分代价，一部分是统计它的子节点的代价，这个只需要将子节点的代价简单相加就可以了，另一个

非卖品，仅供非商业用途或交流学习使用

是对子节点生成的位图做并集的代价，这里 PostgreSQL 数据库给出了一个简单的代价计算模型，就是 BitmapOrPath 的每个子节点需要使用 100.0 × cpu_operator_cost 的代价来做并集操作。另外需要注意的是，BitmapOrPath 中记录的选择率是对子节点的选择率相加，而 BitmapAndPath 中记录的选择率是对子节点的选择率相乘，这些估计都是粗略的，这个选择率主要是给 BitmapHeapPath 做代价估计用的。

BitmapHeapPath 的代价是在 cost_bitmap_heap_path 中完成的，需要注意以下几点：

1）BitmapHeapPath 的启动代价中包含的是它的子节点的总代价，因为它的子节点只负责生成位图，在位图生成之前，是无法获取第一条元组的，因此它的子节点中的代价全部是启动代价。

2）在获取到位图之后会进行堆扫描，因为位图是有序的，所以我们可以假定我们对堆的扫描也是有序的，但这种有序和顺序扫描（SeqScan）的有序是不同的，在顺序扫描（SeqScan）中，我们的扫描过程不只是有序的，而且还是连续的，由于磁盘页面的预取作用，我们可以假定我们的 IO 代价是低的，但位图扫描对堆进行扫描的时候虽然是有序的，但它的“连续性”是不好的，尤其是在选择率低的情况下，很可能在获取下一个页面的时候已经跳过了磁盘预取的范围，因此我们在计算 IO 代价的时候并不是简单地使用 seq_page_cost，而是使用一个估计的值：

```
random_page_cost -(random_page_cost - seq_page_cost) *sqrt(pages_fetched / T)
```

其中 pages_fetched 是要读取的页面数，T 是堆表的总页面数。

6.5　小结

本章主要介绍了 3 种扫描路径：顺序扫描路径、（快速）索引扫描路径、位图扫描路径，并且介绍了这些路径的代价计算方式，实际上除了这 3 种扫描路径外还有 TID 扫描路径、子查询扫描路径等，这些也是常用的扫描路径，有兴趣的读者可以自行研究一下。

本章还介绍了代价的基准单位，PostgreSQL 数据库选择从磁盘顺序读取一个页面的代价作为“单位 1”，这是一个相对代价，它保证了物理路径之间的代价是可以相互比较的。

本章还对并行路径、参数化路径及路径的有序性进行了说明，这些都是提高查询性能的重要手段，需要读者仔细体会它们的含义。

非卖品，仅供非商业用途或交流学习使用

7

第 7 章

动态规划和遗传算法

目前扫描路径已经生成，每个基表都产生了至少一个或者多个扫描路径，这些扫描路径构成了整个查询计划的叶子节点，在查询计划的上层还有连接路径、Sort 路径、物化路径等，这里我们重点关注连接路径，因为连接操作是查询语句最高频的操作。

如果穷举所有的扫描路径来建立所有可能的连接路径，则这个算法的时间复杂度是比较高的，因此选取合适的方法来搜索可能的查询路径就显得非常重要了，在各种最优解的算法中，PostgreSQL 数据库分别选用了动态规划方法和遗传算法来生成连接路径。

动态规划方法需要遍历全部的解空间，它一定能够获得最优解，因此是我们首选的方法，遗传算法则只能尝试从局部最优解向全局最优解不断逼近，但由于遗传代际的数量的限制，最终可能产生的是局部最优解，这种方法在表比较多的时候被采用，因为在表比较多的时候，动态规划的解空间快速地膨胀，可能会导致查询性能的下降，遗传算法的复杂度则可以限制在一定的范围内，下面我们分别来介绍动态规划算法的实现机制和遗传算法的实现机制。

非卖品，仅供非商业用途或交流学习使用

7.1　动态规划

在 deconstruct_jointree 之后，语句中的表被拉平，从树状结构转换成了数组结构，除了内连接，其他类型的连接关系被记录到 SpecialJoinInfo 结构体里，同时被记录到 SpecialJoinInfo 结构体里的还有表之间的连接顺序，根据在第 4 章中介绍的逻辑分解优化阶段对表之间连接顺序的分析，我们可以发现同一个查询语句可能对应多个结果等价的连接顺序，那么这些等价的连接顺序所产生的连接路径的执行代价是否也相同呢？答案是否定的。

假如有 TEST_A、TEST_B、TEST_C，示例如下：

```
INSERT INTO TEST_A SELECT GENERATE_SERIES(1,10000), FLOOR(RANDOM()*100),
FLOOR(RANDOM()*100), FLOOR(RANDOM()*100);
INSERT INTO TEST_B SELECT GENERATE_SERIES(1,10000), FLOOR(RANDOM()*100),
FLOOR(RANDOM()*100), FLOOR(RANDOM()*100);
INSERT INTO TEST_C SELECT GENERATE_SERIES(1,10000), FLOOR(RANDOM()*100),
FLOOR(RANDOM()*100), FLOOR(RANDOM()*100);

postgres=# EXPLAIN (COSTS OFF, ANALYZE ON) SELECT * FROM TEST_A, TEST_B, TEST_C WHERE TEST_A.a >
0 AND TEST_B.a > 5000 AND TEST_C.a > 9999;
                          QUERY PLAN
-----------------------------------------------------------------------------------------
 Nested Loop (actual time=2.888..10862.965 rows=50000000 loops=1)
   ->  Seq Scan on test_a (actual time=0.011..2.290 rows=10000 loops=1)
         Filter: (a > 0)
   ->  Materialize (actual time=0.000..0.283 rows=5000 loops=10000)
         ->  Nested Loop (actual time=2.873..6.033 rows=5000 loops=1)
               ->  Seq Scan on test_c (actual time=2.517..2.517 rows=1 loops=1)
                     Filter: (a > 9999)
                     Rows Removed by Filter: 9999
               ->  Seq Scan on test_b (actual time=0.355..2.569 rows=5000 loops=1)
                     Filter: (a > 5000)
                     Rows Removed by Filter: 5000
 Planning time: 0.141 ms
 Execution time: 13513.525 ms
(13 rows)
```

从示例的执行计划可以看出，TEST_B 和 TEST_C 先做连接产生了 5000 条中间结果，然后将 5000 条中间结果物化，TEST_A 与物化的中间结果再做连接操作。如果交换连接顺序，用 TEST_A 和 TEST_B 先做连接则会产生 50 000 000 条中间结果，如果将 50 000 000 条元组进行物化，则它的代价肯定会高于将 5000 条元组进行物化，因此查询优化器“聪明”地选择了先对

非卖品，仅供非商业用途或交流学习使用

TEST_B 和 TEST_C 做连接，通过这个示例说明了表之间的连接顺序还是非常重要的，因此查询优化器使用动态规划的方法来获得最优的连接顺序。

动态规划方法适用于包含**大量重复子问题**的最优解问题，通过记忆每个子问题的最优解，使相同的子问题只求解一次，下次可以重复利用上次子问题求解的结果，这就要求这些子问题的最优解能够构成整个问题的最优解，也就是要具有**最优子结构的性质（无后效性）**，假如最优的子问题无法构成整个问题的最优解，就无法采用动态规划的方法。

连接树的生成是否符合上面的重复子问题和最优子结构性质？在查询优化求解最优的过程中会生成多条连接树，每个连接树都要计算代价，以如图 7-1 所示的两个连接树为例。

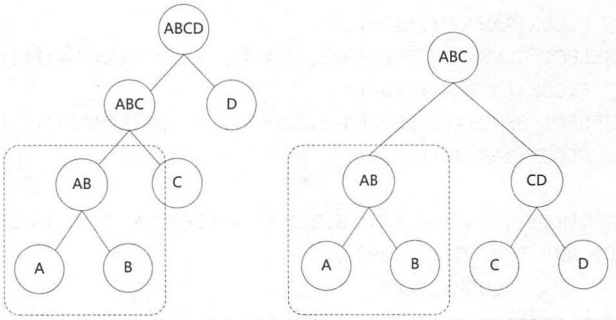


图 7-1　动态规划重复子问题

图 7-1 中的两棵连接树中的 A×B 的连接操作就属于重复子问题，对于多表连接，生成的路径可能有上百棵连接树，这些连接树中的重复子问题的数量就比较可观了，因此连接树的生成具有重复子问题，下面来看一下它是否还满足最优子结构的性质。

对于连接树的每个子问题，通过不断获得“堆积”子问题的最低代价路径，最终迭代式地“堆积”出最终整个连接树的最低代价路径，我们来看一个例子，如图 7-2 所示。

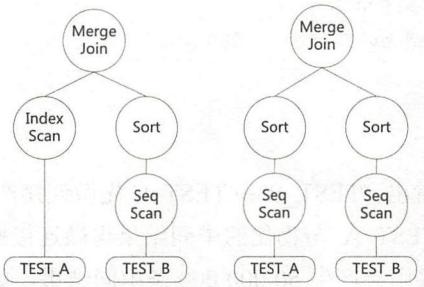


图 7-2　动态规划的适用性

非卖品，仅供非商业用途或交流学习使用

假如有 SQL 语句 SELECT a FROM TEST_A FULL JOIN TEST_B WHERE TEST_A.a = TEST_B.a，并假设 TEST_A.a 是主键，也就是说在 TEST_A.a 属性上有一个主键索引，因此对 TEST_A.a 进行索引扫描产生的扫描结果具有有序的性质。

如果使用动态规划方法来生成执行路径，那么每个路径节点都应该产生一个最优子问题，例如在给 TEST_A 表建立扫描路径的时候，会建立顺序扫描路径和索引扫描路径，但是在这两个路径中哪个是子问题的最优解呢？在介绍索引扫描路径的时候我们有一个感性的认识：如果约束条件的选择率高，通常会选择顺序扫描路径，如果约束条件的选择率低，通常会选择索引扫描路径，然后我们做 3 个假设。

假设 1：约束条件的选择率高，TEST_A 的扫描路径中顺序扫描是最优解。

假设 2：TEST_A 和 TEST_B 的连接操作使用 MergeJoin 方法，如果 TEST_A 采用顺序扫描路径，则还需要显式地对其结果进行排序。

假设 3：TEST_A 的索引扫描路径产生的结果恰好符合 MergeJoin 的 Sort 要求。

那么这时候就可能出现如下情况：

$$顺序扫描代价 < 索引扫描代价$$

$$顺序扫描代价 + 排序代价 > 索引扫描代价$$

这也就导致了对 TEST_A 表的扫描的最优子问题无法产生整个查询计划的最优解的情况。实际上，当对 TEST_A 表生成扫描路径的时候，我们并不知道上层是采用 NestLoopJoin 还是 MergeJoin，也就是说如果在每个 RelOptInfo 中只记录“单一的”“代价最低”的物理路径，则它是不满足最优质结构性质的。

PostgreSQL 数据库对动态规划方法进行了改进，在计算子问题的代价时，不仅要保存最优解，还同时保存较优解，它需要考虑最优的启动代价路径、参数化路径等，这些都记录在 RelOptInfo 之中，也就是说它构成了一个最优解集，在向上“堆积”的过程中可以根据不同的情况，考虑采用解集中的某一个解。

上文中不止一次提到从子问题的最优解“堆积”出整个问题的最优解，动态规划从求解的方式而言通常有递归和迭代两种方法，递归方法通常是不断地划分子问题然后自我调用的过程，迭代则是从子问题直接出发，先求子问题的解，用子问题的解“堆积”整个问题的解，从直观感受上来看，递归是逆向的，迭代是正向的。PostgreSQL 数据库采用的是迭代的方式进行求解，它先尝试求两个表的最优子路径，然后依次迭代成 3 个表、4 个表……依此类推。假设如图 7-3

非卖品，仅供非商业用途或交流学习使用

所示有 A、B、C、D 共 4 个表，则最初 4 个表在一个链表里，这 4 个表已经建立成了 RelOptInfo，并且每个 RelOptInfo 已经建立好了扫描路径，这时的 RelOptInfo 是 RELOPT_BASEREL 类型，通常称之为基表。



图 7-3　动态规划堆积过程，第 1 层

动态规划方法首先考虑两个表的连接，其中优先考虑有连接关系（SpecialJoinInfo）的表进行连接，两个表的连接可以建立一个新的 RelOptInfo 标识，生成的连接路径也记录到这个 RelOptInfo 中，这时的 RelOptInfo 是 RELOPT_JOINREL 类型的，表示它是连接操作产生的，它里面存储的也是连接路径，如图 7-4 所示。

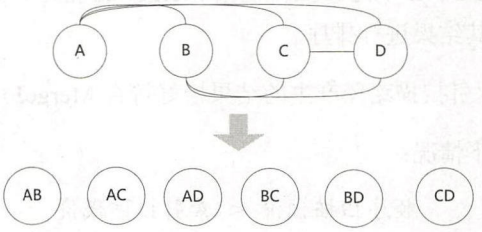


图 7-4　动态规划堆积过程，第 2 层

将基于两个表连接的 RelOptInfo 和最初的 4 个基表进行连接，可以生成基于 3 个表连接的新的 RelOptInfo，这样就又向前推进了一层，如图 7-5 所示（虽然图中的连线看似非常密集，但是由于有连接顺序的限制、由于动态规划方法记录了重复的子问题、连接路径建立过程中的优胜劣汰等原因，有些连线是不会产生的，因此算法的时间复杂度是可控的）。

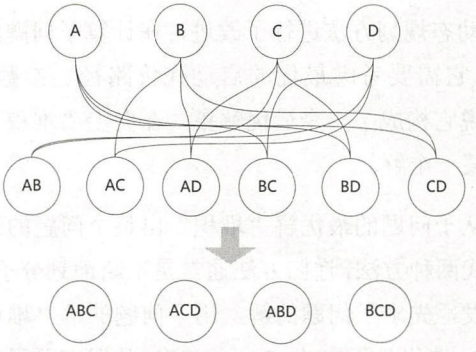


图 7-5　动态规划堆积过程，第 3 层

非卖品，仅供非商业用途或交流学习使用

然后用基于 3 个表连接的 RelOptInfo 和最初的 4 个基表进行连接，最终生成整个问题的最优路径，如图 7-6 所示。

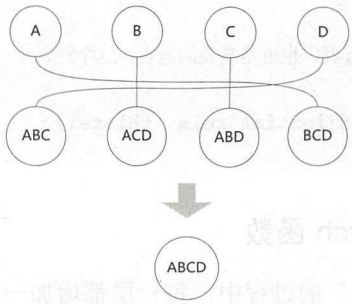


图 7-6　动态规划堆积过程，第 4 层

7.1.1　make_rel_from_joinlist 函数

动态规划的实现代码在 make_rel_from_joinlist 函数中，它的输入参数是被 deconstruct_jointree 函数拉平之后的 RangeTableRef 链表（需要注意的是这个链表不一定是完全被拉平的，由于受到 from_collapse_limit 参数和 join_collapse_limit 参数的影响，可能有些子连接树没有被拉平，对于没有被拉平的子连接树，会递归调用 make_rel_from_joinlist 函数处理），每个 RangeTableRef 代表一个表，然后就可以根据这个链表来查找基表的 RelOptInfo，查找到的基表 RelOptInfo 就能构成第一个初始链表，也就是每个表的扫描路径的最优解集。

```
//遍历拉平之后的 joinlist，这个链表是 RangeTableRef 的链表
foreach(jl, joinlist)
{
    Node   *jlnode = (Node *) lfirst(jl);
    RelOptInfo *thisrel;

    //多数情况下都是 RangeTableRef，根据 RangeTableRef 中存放的下标值（rtindex）
    //查找对应的 RelOptInfo
    if (IsA(jlnode, RangeTblRef))
    {
        int          varno = ((RangeTblRef *) jlnode)->rtindex;
        thisrel = find_base_rel(root, varno);
    }
    //受到 from_collapse_limit 参数和 join_collapse_limit 参数的影响，
    //也存在没有拉平的节点，这里递归调用 make_rel_from_joinlist 函数
    else if (IsA(jlnode, List))
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
            thisrel = make_rel_from_joinlist(root, (List *) jlnode);
        else
            elog(……);

        //这里就生成了第一个初始链表，也就是基表的链表，这个链表是
        //动态规划方法的基础
        initial_rels = lappend(initial_rels, thisrel);
    }
```

7.1.2 standard_join_search 函数

由于动态规划方法在“堆积”的过程中，每一层都增加一个表，当所有的表都增加完毕的时候，连接树也就生成了，因此需要“堆积”的层数就是表的数量，也就是说有 N 个表，动态规划方法就要堆积 N 次，即需要通过 N 层的迭代才能生成最后的连接树（假如基表的链表生成不算在内的话，则准确地说是 N-1 次）。

我们创建一个“链表的数组”，数组中的每个链表用来存放动态规划方法中一层的所有 RelOptInfo，数组的第一个链表存放的是基表的链表。

```
//创建一个链表的链表，为了后面使用方便，这里多创建一个，
//把基表的链表放到了下标为 1 的位置，也就是链表的第 2 个位置，
//这样 root->join_rel_level[0]永远为 NULL，不会被用到
root->join_rel_level = (List **) palloc0((levels_needed + 1) * sizeof(List *));
root->join_rel_level[1] = initial_rels;
```

做好了初始工作，就可以开始尝试构建上一层的 RelOptInfo。

```
    //第 1 层已经初始化好，从第 2 层开始生成
    for (lev = 2; lev <= levels_needed; lev++)
    {
        //在 join_search_one_level 函数中生成对应的 lev 层的所有的 RelOptInfo
        join_search_one_level(root, lev);

        ……//省略部分代码
    }
```

7.1.3 join_search_one_level 函数

join_search_one_level 函数的输入参数有两个，如表 7-1 所示，它们分别存储了之前生成的 RelOptInfo 集合和当前需要生成连接操作的层次。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

表 7-1　join_search_one_level函数的输入参数

参数名	参数类型	描述
root	[IN] PlannerInfo *	包含了join_rel_level这个链表的链表，保存了目前生成的所有子连接树集合
level	[IN] int	这次要生成第几层的子连接树集合，这时候level-1之下的其他层的子连接树集合已经生成完毕了

为了生成当前层次的 RelOptInfo，join_search_one_level 函数做了 3 次“努力”，如图 7-7 所示。

- 尝试生成左深树和右深树。
- 尝试生成浓密树。
- 尝试生成基于卡氏积的连接路径。

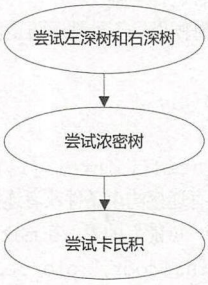


图 7-7 连接路径做了 3 次努力

7.1.3.1　左深树和右深树

左深树和右深树是同时生成的，因为在生成连接路径的时候，populate_joinrel_with_paths 函数会尝试将两个 RelOptInfo 的位置调换。如图 7-8 所示，待连接的是一个子树和一个基表，它们通过变换顺序就可以分别生成左深树和右深树。

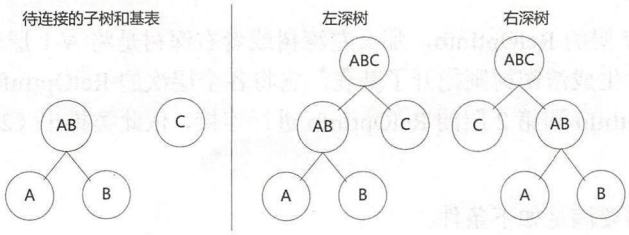


图 7-8　左深树和右深树

版权所有，严禁传播，违者自负法律责任！


```
//对当前层的上一层进行遍历，也就是说如果要生成第 4 层的 RelOptInfo，
//就要取第 3 层的 RelOptInfo 和第 1 层的基表尝试做连接
foreach(r, joinrels[level - 1])
 {
     RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);
     //如果两个 RelOptInfo 之间与连接关系或者有连接顺序的限制
     //那么优先给这两个 RelOptInfo 生成连接
     // has_join_restriction 函数可能误判，不过后续还会有更精细的筛查
     if (old_rel->joininfo != NIL || old_rel->has_eclass_joins ||
         has_join_restriction(root, old_rel))
     {
         ListCell   *other_rels;
         //要生成第 N 层的 RelOptInfo，就需要第 N-1 层的 RelOptInfo 和
         //第 1 层的基表集合进行连接
         //即：如果要生成第 2 层的连接树子集，那么就变成第 1 层的
         //基表集集和第 1 层的基表集合进行连接
         //因此，需要对生成第 2 层的时候做一下处理，防止自己和自己连接的情况
         if (level == 2)
             other_rels = lnext(r);
         else
             other_rels = list_head(joinrels[1]);
         //old_rel “可能”和其他表有连接约束条件或者连接顺序限制
         //other_rels 里面就是那些“可能”的表，在 make_rels_clause_joins 函数里会精确地判断
         make_rels_by_clause_joins(root,  old_rel,  other_rels);
     }
     else
     {
         //对没有连接关系的表或连接顺序限制的表也需要尝试生成连接路径
         make_rels_by_clauseless_joins(root,  old_rel,  list_head(joinrels[1]));
     }
 }
```

7.1.3.2 浓密树

假设要生成第 *N* 层的 RelOptInfo，那么左深树或者右深树是将 *N*-1 层的 RelOptInfo 和第 1 层的基表进行连接；生成浓密树则抛开了基表，它将各个层次的 RelOptInfo 尝试进行连接，例如将第 *N*-2 层 RelOptInfo 和第 2 层的 RelOptInfo 进行连接，依此类推出（2，*N*-2）、(3，*N*-3)、(4, *N*-4)等多种情况。

要建立浓密树需要满足如下条件。

- 两个 RelOptInfo 需要有相关的约束条件（连接条件）或者有连接顺序的限制。

版权所有，严禁传播，违者自负法律责任！

- 两个 RelOptInfo 分别所代表的表不能有交集。

```
for (k = 2;; k++)
{
    int           other_level = level - k;
    foreach(r, joinrels[k])
    {
        //有连接条件或者有连接顺序的限制
        if (old_rel->joininfo == NIL && !old_rel->has_eclass_joins &&
            !has_join_restriction(root, old_rel))
            continue;
        ......
        for_each_cell(r2, other_rels)
        {
            RelOptInfo *new_rel = (RelOptInfo *) lfirst(r2);
            //不能有交集
            if (!bms_overlap(old_rel->relids, new_rel->relids))
            {
                //有相关的连接条件或者有连接顺序的限制
                if (have_relevant_joinclause(root, old_rel, new_rel) ||
                    have_join_order_restriction(root, old_rel, new_rel))
                {
                    (void) make_join_rel(root, old_rel, new_rel);
                }
            }
        }
    }
}
```

7.1.3.3　卡氏积

卡氏积是连接中的“大饼卷一切”，这也是最后的尝试，如果仍然无法生成连接关系，那么 SQL 语句就要终止，在 PostgreSQL 数据库的注释中给出了必须通过卡氏积才能实现的的例子：

```
--这里假设 a INNER JOIN b 没有被拉平（没被拉平的原因请参考 make_rel_from_joinlist 函数）
--另外要求 a 和 b 与上层的表有连接关系
SELECT ... FROM a INNER JOIN b ON TRUE, c, d, ... WHERE a.w = c.x and b.y = d.z;
```

7.2　遗传算法

物竞天择，适者生存，这是大自然的普遍规律，遗传算法正是模拟这种规律，通过基因的

版权所有，严禁传播，违者自负法律责任！

排列组合产生新的染色体，再通过染色体的杂交和变异获得下一代染色体，并且建立适应性函数对染色体进行筛选，淘汰掉不良的染色体，保留优秀的染色体，通过多次的代际遗传，使得问题的解从局部最优解向全局最优解不断地推进。

动态规划方法是通过使用独立子问题逐步推进到解决整个问题的；遗传算法则是一个选择的过程，它通过将染色体杂交构建新染色体的方法增大解空间，并在解空间中随时通过适应度函数进行筛选，推举良好的基因，也淘汰掉不良的基因。动态规划获得的解一定是全局最优解，遗传算法最终不一定能达到全局最优解，但我们可以通过改进杂交和变异的方式，来争取尽量地靠近全局最优解。

在 PostgreSQL 数据库中，遗传算法是动态规划方法的有益补充，只有在 enable_geqo 打开并且待连接的 RelOptInfo 的数量超过 geqo_threshold（默认为 12 个）的情况下，才会使用遗传算法。

遗传算法的实现步骤如下。

- 种群初始化：对基因进行编码，并通过对基因进行随机的排列组合，生成多个染色体，这些染色体构成一个新的种群，另外，在生成染色体的过程中同时计算染色体的适应度。
- 选择染色体：通过随机选择（实际上通过基于概率的随机数生成算法，这样能倾向于选择出优秀的染色体），选择出用于交叉和变异的染色体。
- 交叉操作：染色体进行交叉，产生新的染色体并加入种群。
- 变异操作：对染色体进行变异操作，产生新的染色体并加入种群。
- 适应度计算：对不良的染色体进行淘汰。

如果用遗传算法解决货郎问题（TSP），则可以将城市作为基因，走遍各个城市的路径作为染色体，路径的总长度作为适应度，适应度函数负责筛选掉比较长的路径，保留较短的路径，算法的步骤如下。

- 对各个城市进行编号，将各个城市根据编号进行排列组合，生成多条新的路径（染色体），然后根据各城市间的距离计算整体路径长度（适应度），多条新路径构成一个种群。
- 选择两个路径进行交叉（需要注意在交叉生成的新染色体中不能重复出现同一个城市），对交叉操作产生的新路径计算路径长度。
- 随机选择染色体进行变异（通常方法是交换城市在路径中的位置），对变异操作后的新路径计算路径长度。
- 对种群中所有路径进行基于路径长度由小到大排序，淘汰掉排名靠后的路径。

PostgreSQL 数据库的遗传算法正是模拟了解决货郎问题的方法，将 RelOptInfo 作为基因，

版权所有，严禁传播，违者自负法律责任！

将最终生成的连接树作为染色体，将连接树的总代价作为适应度；适应度函数则是基于路径的代价进行筛选。但是 PostgreSQL 数据库的连接路径的搜索和货郎问题的路径搜索略有不同，货郎问题不存在路径不通的问题，两个城市之间是相通的，可以计算任意两个城市之间的距离，而在数据库中由于连接条件的限制，可能两个表无法正常连接，或者整个连接树都无法生成。另外需要注意的是，PostgreSQL 数据库的基因算法的实现方式和通常的遗传算法略有不同，在于其没有变异的过程，只通过交叉产生新的染色体。

为了方便对代码进行解读，我们将打开遗传算法的边界条件调低（即 RelOptInfo 数量没有达到 geqo_threshold 的限制，也尝试使用遗传算法），这不会影响我们对代码的解读。在解读代码的过程中以 TEST_A、TEST_B、TEST_C、TEST_D 这 4 个表为例。

遗传算法的总入口是 geqo 函数，输入参数为 root（查询优化的上下文信息）、number_of_rels（要进行连接的 RelOptInfo 的数量）、initial_rels（所有的基表）。

7.2.1　种群初始化

RelOptInfo 作为遗传算法的基因，首先需要进行基因编码，PostgreSQL 数据库采用实数编码的方式，也就是用{1,2,3,4}分别代表 TEST_A、TEST_B、TEST_C、TEST_D 这 4 个表。

然后通过 gimme_pool_size 函数来获得种群的大小，种群的大小受 geqo_pool_size 和 geqo_effort 两个参数的影响，种群用 Pool 结构体进行表示，染色体用 Chromosome 结构体来表示。

```
//染色体表示
typedef struct Chromosome
{
    // string 实际是一个整型数组，它代表基因的一种排序方式，也就对应一棵连接树
    //例如{1,2,3,4}对应的就是 TEST_A JOIN TEST_B JOIN TEST_C JOIN TEST_D
    //例如{2,3,1,4}对应的就是 TEST_B JOIN TEST_C JOIN TEST_A JOIN TEST_D
    Gene    *string;
    Cost     worth;      //染色体的适应度，实际上就是路径代价
} Chromosome;

//种群的表示
typedef struct Pool
{
    Chromosome *data;. //染色体数组，数组中的每个元素都是一个连接树
    int           size;  //染色体的数量，即 data 中连接树的数量，由 gimme_pool_size 生成
```

版权所有，严禁传播，违者自负法律责任！


```
    int             string_length;  //每个染色体中的基因数量，和基表的数量相同
} Pool;
```

另外，通过 gimme_number_generations 函数来获取染色体交叉的次数，交叉的次数越多则产生的新染色体也就越多，也就更可能找到更好的解，但是交叉次数多也影响性能，用户可以通过 geqo_generations 参数来调整交叉的次数。

目前已经确定的变量有：

- 通过 gimme_pool_size 确定的染色体的数量（Pool.size）。
- 每个染色体中基因的数量（Pool.string_length）和基表的数量相同。

然后就可以开始生成染色体，染色体的生成采用的是改进的 Fisher-Yates 洗牌算法，最终生成 Pool.size 条染色体。PostgreSQL 数据库的早期版本使用经典的 Fisher_Yates 洗牌算法来生成染色体，流程是按序初始化染色体，然后通过随机数随机交换，最终产生染色体的随机序列，目前的改进主要是结合了实数编码有序的特点，将初始化染色体和随机交换结合起来一起进行，具体的算法实现在 init_tour 函数中，生成一条染色体的流程如表 7-2 所示。

表 7-2　染色体流程流程

轮数 i	结果集 tour	随机数 范围	随机数 j	说明
初值	1			给第1个基因赋值为编码1
1	2 1	0～1	0	轮数1 != 随机数0，因此调整tour[0]到tour[1]的位置，tour[0] = 1 + 1 = 2
2	2 1 3	0～2	2	轮数2 == 随机数2，tour[2] = 2 + 1 = 3
3	2 4 3 1	0～3	1	轮数3 != 随机数1，调整tour[1]到tour[3]的位置，tour[1] = 3 + 1 = 4

假设在 Pool 种群中共有 4 条染色体，用图来描述其结构，如图 7-9 所示。

然后对每条染色体计算适应度（worth），计算适应度实际上就是根据染色体的基因编码顺序产生连接树并对连接树求代价的过程。

在早期的 PostgreSQL 数据库中，每个染色体都默认使用的是左深树，因此在每个染色体的基因编码确定后，它的连接树也就随之确定了，比如针对{2, 4, 3, 1}这样一条染色体，它对应的连接树就是（（（TEST_B, TEST_D）, TEST_C）, TEST_A），如图 7-10 所示。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

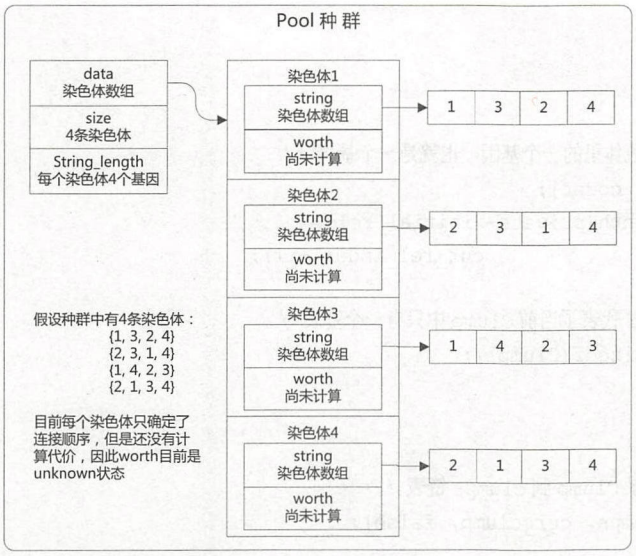


图 7-9　种群初始化的内存结构

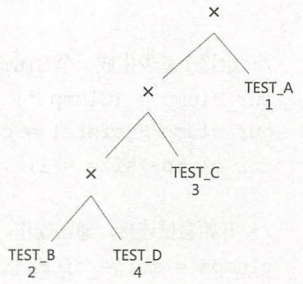


图 7-10　每个染色体代表的连接树

但左深树不一定是合法的连接路径，在第 4 章中对于不同的逻辑连接类型都给出了等价的连接顺序交换的等式，这里的染色体是随机编码的，就有可能有些染色体对应的连接路径是不合法的，这就导致会生成很多无效的染色体，PostgreSQL 数据库早期的版本在生成不合法的路径之后就会报错退出，导致查询无法正常进行。在最新的 PostgreSQL 数据库中对这种情况有所改进，给出了新的算法用来生成连接树，左深树不再是唯一选择，还会尝试其他类型的连接树，这就增加了生成连接树的可能；另外，即使没有生成合法的连接树，也不再直接报错退出事务，而是将这条染色体的适应度的值（worth）修改为 DBL_MAX 值，使这个连接树的代价无穷大，这样这条染色体（连接树）在根据适应度筛选时就会被淘汰掉。

PostgreSQL 数据库通过 geqo_eval 函数来生成计算适应度，它首先根据染色体的基因编码生成一棵连接树，然后计算这棵连接树的代价。

遗传算法使用 gimme_tree 函数来生成连接树，而 gimme_tree 则递归调用了 merge_clump，merge_clump 的作用是将能够进行连接的表尽量连接并且生成连接子树，并记录每个连接子树中节点的个数，然后将连接子树记录到 clumps 链表，而且按照节点个数从高到低的顺序记录到 clumps 链表。

```
//循环遍历所有的表，尽量将能连接的表连接起来
for (rel_count = 0; rel_count < num_gene; rel_count++)
{
```


严禁网络传播本PDF，违者责任自负！

```
    int          cur_rel_index;
    RelOptInfo *cur_rel;
    Clump       *cur_clump;

    /* tour 代表一条染色体，这里获取染色体里的一个基因，也就是一个基表 */
    cur_rel_index = (int) tour[rel_count];
    cur_rel = (RelOptInfo *) list_nth(private->initial_rels,
                                      cur_rel_index - 1);

    /* 给这个基表生成一个 Clump，size=1 代表了当前 Clump 中只有一个基表*/
    cur_clump = (Clump *) palloc(sizeof(Clump));
    cur_clump->joinrel = cur_rel;
    cur_clump->size = 1;

    /* 开始尝试连接，递归操作，并负责记录 Clump 到 clumps 链表 */
    clumps = merge_clump(root, clumps, cur_clump, false);
}
```

我们再以{2, 4, 3, 1}这样一条染色体为例看一下连接树生成的过程，这里假定：

- 2 和 4 不能连接。
- 4 和 3 能够连接。
- 2 和 1 能够连接。

连接树生成的过程如表 7-3 所示。

表 7-3　连接树生成的新方法

轮数 relcount	连接结果集 clumps	说明
初始	NULL	创建基因为2的节点cur_clump，cur_clump.size = 1
0	{2}	因为clumps == NULL，cur_clump没有连接表，将cur_clump直接加入到clumps
1	{2}，{4}	创建基因为4的节点cur_clump，cur_clump.size = 1 将基因为4的cur_clump和clumps链表的里的节点尝试连接 因为2和4不能连接，节点4也被加入到clumps

严禁网络传播本PDF，违者责任自负！

续表

轮数 relcount	连接结果集 clumps	说明
2	{2}	创建基因为3的节点cur_clump，cur_clump.size = 1 遍历clumps链表，分别尝试和2、4进行连接 发现和4能进行连接 创建基于3和4的连接的新old_clumps节点，ols_clumps.size = 2，在clumps链表中删除节点4
	{3, 4} {2}	用2和4连接生成的新的old_clumps作为参数递归调用merge_clump 用old_clumps和clumps链表里的节点再尝试连接 发现不能连接（即{3,4}和{2}不能连接），那么将old_clumps加入clumps，因为old_clumps.size 目前最大，插入到clumps最前面
3	{3, 4}	创建基因为1的节点cur_clump，cur_clump.size = 1 遍历clumps链表，分别尝试和{3,4}、{2}进行连接 发现和2能进行连接 创建基于1和2的新old_clumps节点， ols_clumps.size = 2，在clumps链表中删除节点2
	{3, 4} {1, 2}	用1和2连接生成的新的old_clumps作为参数递归调用merge_clump 用old_clumps和clumps链表里的节点尝试连接 发现不能连接，将old_clumps加入clumps 因为old_clumps.size = 2，插入clumps最后

结合例子中的步骤，可以看出 merge_clumps 函数的流程就是不断地尝试生成更大的 clump。

```
//如果能够生成连接，则通过递归尝试生成节点数更多的连接
if (joinrel)
{
.....
//生成新的连接节点，连接的节点数增加
    old_clump->size += new_clump->size;
    pfree(new_clump);

    /* 把参与了连接的节点从 clumps 链表里删除 */
    clumps = list_delete_cell(clumps, lc, prev);
  // 以 clumps 和新生成的连接节点（old_clump）为参数，继续尝试生成连接
    return merge_clump(root, clumps, old_clump, force);
}
```


严禁网络传播本PDF，违者责任自负！

根据表 7-3 中的示例，最终在 clumps 链表中有两个节点，分别是两棵连接子树，然后我们将 force 设置成 true 后，再次尝试连接这两个节点。

```
//在 clumps 中有多个节点，证明连接树没有生成成功
if (list_length(clumps) > 1)
{
    ......
    foreach(lc, clumps)
    {
        Clump       *clump = (Clump *) lfirst(lc);
        //设置 force 参数为 true，尝试无条件连接
        fclumps = merge_clump(root, fclumps, clump, true);
    }
    clumps = fclumps;
}
```

对于不能生成有效连接树的染色体，设置它的适应度为 DBL_MAX，PostgreSQL 数据库遗传算法的适应度函数就是按照代价的大小对染色体进行排序（快速排序），这样代价小的染色体就排在前面，代价高的染色体就排在后面，这样就可以把排名靠后的染色体淘汰掉。

7.2.2 选择算子

在种群生成之后，就可以进行代际遗传优化，从种群中随机选择两个染色做交叉操作，这样就能产生一个新的染色体。

由于种群中的染色体已经按照适应度排序了，对我们来说适应度越低（代价越低）的染色体越好，因此选择操作的“随机选择”并不是真正的随机，而是基于概率分布的随机。这样在选择父亲染色体和母亲染色体的时候更倾向于选择适应度低的染色体。

```
//父亲染色体和母亲染色体通过 linear_rand 函数选择
first = linear_rand(root, pool->size, bias);
second = linear_rand(root, pool->size, bias);
```

要生成基于某种概率分布的随机数，需要首先知道概率分布函数或概率密度函数，PostgreSQL 数据库给出的概率密度函数为（代码注释是“概率分布函数（probability distribution function）”，但根据源代码实际内容推测，这里应该是“概率密度函数（probability density function）”）：

$$f(x) = bias - 2(bias - 1)x \quad (0 < x < 1);$$

通过概率密度函数获得概率分布函数：

严禁网络传播本PDF，违者责任自负！

$$F(x) = \int_{-\infty}^{x} f(x)dx = \begin{cases} 0, & x \le 0 \\ bias\ x - (bias-1)x^2, & 0 < x < 1 \\ 1, & x \ge 1 \end{cases}$$

然后通过概率分布函数根据逆函数法可以获得符合概率分布的随机数，对函数

$$F(x) = bias\ x - (1 - bias)x^2$$

求逆函数

$$F^{-1}(x) = \frac{bias - \sqrt{bias^2 - 4(bias-1)\ y}}{2(bias-1)}$$

这和源代码中 linear_rand 函数的实现是一致的。

```
//先求√(bias² − 4(bias − 1) y)的值
double        sqrtval;
sqrtval = (bias * bias) - 4.0 * (bias - 1.0) * geqo_rand(root);
if (sqrtval > 0.0)
    sqrtval = sqrt(sqrtval);

//然后计算(bias−√(bias²−sqrtval))/(2(bias−1))的值，其为基于概率分布随机数且符合[0,1]分布
//max 是种群中染色体的数量
//index 就是满足概率分布的随机数，且随机数的值域在[0, max]
index = max * (bias - sqrtval) / 2.0 / (bias - 1.0);
```

我们把基于概率的随机数生成算法的代码提取出来单独形成一个小程序，看一下它生成随机数的特点。设 $bias = 2.0$（则概率密度函数为 $2 - 2x$），然后用小程序生成随机数，按照对算法的理解，生成的随机数是 0～1 的符合 $2 - 2x$ 分布的随机数，共生成 10000 个，然后以 0.1 为单位统计这 10000 个随机数落到了哪个区间，比如随机数 0.05 就落到了 0.0～0.1 的区间，而随机数 0.45 就落到了 0.4～0.5 的区间。

然后依据概率密度函数计算各个区间的理论概率值并进行对比，比如对于 0.4～0.5 的区间我们计算其理论概率如下：

$$F(x) = \int_{0.4}^{0.5} f(x)dx = 0.11 = 11\%$$

理论概率和小程序生成的随机概率对比如表 7-4 所示。

严禁网络传播本PDF，违者责任自负！

表 7-4　基于概率的随机数示例

分段（区间）	总数（个）	算法概率（%）	理论概率（%）
0.0 -0. 10	1923	19.23%	19.00%
0.10 – 0.20	1671	16.71%	17.00%
0.20 – 0.30	1502	15.02%	15.00%
0.30 – 0.40	1315	13.15%	13.00%
0.40 – 0.50	1075	10.75%	11.00%
0.50 – 0.60	913	9.13%	9.00%
0.60 – 0.70	731	7.31%	7.00%
0.70 – 0.80	494	4.94%	5.00%
0.80 – 0.90	277	2.77%	3.00%
0.90 – 1.00	99	0.99%	1.00%

从表 7-4 中可以看出无论是随机概率还是理论概率都基本是吻合的，并且可以看出它们的数值是依次下降的，也就是说在选择父母染色体的时候更倾向于选择适应度更低（代价更低）的染色体。

7.2.3　交叉算子

通过选择算子选择出父母染色体之后，则可以对选出的父母染色体进行交叉操作，生成新的子代染色体。

PostgreSQL 提供了多种交叉方法，包括基于边的重组交叉方法、部分匹配交叉方法、循环交叉、位置交叉、顺序交叉等。在源代码分析的过程中，我们以位置（PX）交叉方法为例进行说明，因为这种交叉方法简单明了，易于讲解，而且不会影响我们理解遗传算法生成连接树的过程。

假如选择父染色体的基因编码为{1, 3, 2, 4}，适应度为 100，母染色体的基因编码为{2, 3, 1, 4}，适应度为 200，这时子染色体还没有生成，如图 7-11 所示。

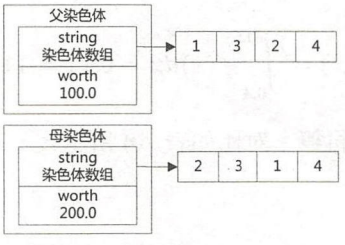


图 7-11　父母染色体的内容

严禁网络传播本PDF，违者责任自负！

交叉操作需要生成一个随机数 num_positions，这个随机数的位置介于基因总数的 1/3～2/3 的位置，这个随机数代表了有多少父染色体的基因要按位置遗传给子染色体。

```
//num_positions 决定了父染色体遗传多少基因给子染色体
num_positions = geqo_randint(root, 2 * num_gene / 3, num_gene / 3);

/* choose random position */
for (i = 0; i < num_positions; i++)
{
    //随机生成位置，将父染色体这个位置的基因遗传给子染色体
    pos = geqo_randint(root, num_gene - 1, 0);

    offspring[pos] = tour1[pos];    /* transfer cities to child */
    //标记这个基因已经使用，母染色体不能再遗传相同的基因给子染色体
    city_table[(int) tour1[pos]].used = 1;   /* mark city used */
}
```

假设父染色体需要遗传两个基因给子染色体，分别传递第 1 号基因和第 2 号基因，那么子染色体目前的状态如图 7-12 所示。

目前子染色体已经有了 3 和 2 两个基因，则母染色体排除这两个基因后，还剩下 1 和 4 两个基因，将这两个基因按照母染色体中的顺序写入子染色体中，新的子染色体就生成了，如图 7-13 所示。

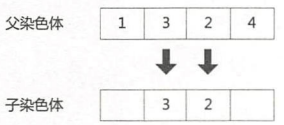


图 7-12　父染色体的遗传方式

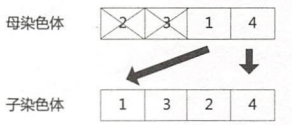


图 7-13　母染色体的遗传方式

7.2.4　适应度计算

新生成的子染色体要通过 geqo_eval 来计算适应度，并且使用 spread_chromo 函数将染色体加入到种群中。

由于种群中的染色体需要保持有序的状态，spread_chromo 函数会使用二分法遍历种群中的染色体，并且比较种群中的染色体和新染色体的适应度大小，查找新染色体的插入位置，排在它后面的染色体自动退后一格，最后一个染色体被淘汰，如果新染色体的适应度最大（代价最大），那么直接被淘汰。

通过选择优秀的染色体、多次的代际交叉，会不断地更新种群的染色体，也就达到了不断地从局部最优解向全局最优解逼近的目标。

7.3 小结

本章重点介绍了连接路径的搜索算法，我们在第1章中介绍过连接路径的搜索算法有3种：自底向上的方法、自顶向下的方法、随机搜索方法，从本章的介绍可以看出，动态规划强调的是“堆积”的过程，因此它采用的是自底向上的方法，而遗传算法借用了洗牌算法、基于概率的随机数生成算法等，强调了染色体生成的随机性，因此它属于随机搜索算法。

在路径搜索的过程中，还借用了一些启发式规则来对搜索空间的规模进行限制，例如在生成连接路径时优先为有连接条件的两个表生成连接路径，这样能保证连接条件尽早地被使用，也就能尽早地过滤数据。

8

第 8 章

连接路径

在前面的章节中已经介绍了针对每个基表的扫描路径，也介绍了生成连接路径时所使用的算法——动态规划方法和遗传算法，在分析动态规划方法阶段只把对源代码的解读推进到 join_search_one_level 函数中，在分析遗传算法的阶段则把对源代码的解读推进到 merge_clump 函数中，物理优化的部分从这两个函数开始就进入到了建立连接路径的阶段。

连接路径指的是物理连接路径，也就是通过这种路径来实现逻辑连接操作（在 SQL 查询语句中指定的 InnerJoin、LeftJoin 等连接方法是连接的逻辑运算，和物理路径是不同的，数据库使用物理连接路径来实现逻辑连接运算），通常而言，连接操作的实现有 3 个经典的方法，它们分别是 NestloopJoin、HashJoin 和 MergeJoin，建立连接路径的过程就是不断地尝试生成这 3 种路径的过程。

相同的两个基表（RelOptInfo）要建立连接关系，由于它们采用的物理路径不同，对应的路径的代价也就不同，因此在建立连接路径的过程中，需要不断地尝试对路径进行筛选，尽早地淘汰掉一些明显比较差的路径，从时间和空间上减少查询优化器的时间消耗。

8.1 检查

在动态规划方法中需要将*N*-1层的每个RelOptInfo和第1层的每个RelOptInfo尝试做连接，将连接产生的新的RelOptInfo保存到第*N*层，这个算法的时间复杂度是*O*(*M***N*)，在*M*和*N*数值比较大的情况下，它的解空间也会膨胀得比较快，算法的性能不高，从而影响搜索最优解的效率，但是在两个RelOptInfo做尝试连接时，有些是明显不适合做连接操作的，可以通过检查的方式避开在这些RelOptInfo之间做连接操作，也就是说可以通过增加剪枝函数来避免进行无效的搜索。

在SQL语句中会通过逻辑连接操作符和约束条件来指定两个表之间的逻辑连接关系，其中约束条件能起到过滤连接结果的作用，因此如果两个表上有约束条件，那么我们就可以尝试优先对这两个表先做连接，这样就能保证有约束条件的表在查询计划的下层，也就是说能尽早地过滤数据，减少查询计划上层节点的计算量。

另外，两个表之间可能会有连接顺序的限制，比如在SQL查询语句中指定Lateral语义就能导致两个表的连接必须有先有后，在建立物理连接路径的时候也必须考虑这种先后关系。

8.1.1 初步检查

在join_search_one_level函数中，我们对第*N*-1层的（也就是join_search_one_leve函数中的level－1层）每个RelOptInfo进行初步的分析，它如果满足以下几个条件之一：

- RelOptInfo->joininfo不是NULL，证明当前的RelOptInfo有和其他RelOptInfo相关的约束条件，也就是说这个RelOptInfo可能和其他表有连接关系。
- RelOptInfo->has_eclass_joins的值是true，表明了在等价类的记录中有当前RelOptInfo和其他RelOptInfo可能存在等值连接条件。
- has_join_restriction函数的返回值是true，说明当前的RelOptInfo和其他RelOptInfo有连接顺序的限制。

那么这个RelOptInfo和第1层的RelOptInfo尝试做连接操作的时候就进入到make_rels_by_clause_joins函数，否则进入到make_rels_by_clauseless_joins函数，从函数名可以看出来一个是基于连接条件来建立连接路径的，另一个是在没有连接条件的情况下尝试生成基于卡氏积的连接路径的。

我们再分析一下has_join_restriction函数的源码，它用来判断第*N*-1层的这个RelOptInfo是否和其他RelOptInfo有连接顺序的限制。

```
static bool has_join_restriction(PlannerInfo *root, RelOptInfo *rel)
{
   ListCell   *l;

   //如果这个 RelOptInfo 涉及了 Lateral，那么一定是有连接顺序的限制了
   if (rel->lateral_relids != NULL || rel->lateral_referencers != NULL)
      return true;

   // PlaceHolderVar 必须在下层求值，也就是说有些 RelOptInfo 必须处于下层
   foreach(l, root->placeholder_list)
      ……

   //join_info_list 里面保存的是除 InnerJoin 外的连接关系，
   foreach(l, root->join_info_list)
   {
      SpecialJoinInfo *sjinfo = (SpecialJoinInfo *) lfirst(l);

      /* ignore full joins --- other mechanisms preserve their ordering */
      if (sjinfo->jointype == JOIN_FULL)
         continue;

      //这个 SpecialJoinInfo 已经被 RelOptInfo 包含了，跳过这个 SpecialJoinInfo
      //例如有连接关系 A LJ (B LJ C)，假如当前的 RelOptInfo(即 rel)代表的是{B,C}
      //如果当前的 SpecialJoinInfo(即 sjinfo)也是(B LJ C)的时候，就跳过这个 SpecialJoinInfo
      if (bms_is_subset(sjinfo->min_lefthand, rel->relids) &&
         bms_is_subset(sjinfo->min_righthand, rel->relids))
         continue;

      //min_lefthand 和 min_righthand 对连接顺序有控制的作用，
      //如果 RelOptInfo 的 relids 和其中任一个有交集，
      //它可能就涉及连接顺序的限制
      if (bms_overlap(sjinfo->min_lefthand, rel->relids) ||
         bms_overlap(sjinfo->min_righthand, rel->relids))
         return true;
   }

   return false;
}
```

初步检查只是基于单个 RelOptInfo 的判断，它是一种“可能性”的判断，基于当前的 RelOptInfo 判断它是否和其他 RelOptInfo 有连接条件或者连接顺序限制的可能性，当有了这种可能性时，就进入精确检查阶段。

8.1.2 精确检查

在进行了初步的判断后，如果没有判断出这个 RelOptInfo 和其他 RelOptInfo 有连接条件或者连接顺序的限制，那么就进入 make_rels_by_clauseless_joins 函数中，将当前的 RelOptInfo 和第 1 层的所有 RelOptInfo 尝试建立连接路径，但是如果判断当前的 RelOptInfo 和其他 RelOptInfo 有“可能”有连接条件或者连接顺序的限制，那么就进入 make_rels_by_clause_joins 函数中，它会从第 1 层的所有 RelOptInfo 中筛选出真正和当前 RelOptInfo 有连接条件或者有连接顺序限制的 RelOptInfo。

have_relevant_joinclause 函数负责筛选有连接条件的 RelOptInfo，have_join_order_restriction 函数负责判断两个 RelOptInfo 之间是否有连接顺序的限制，这两个函数判断的是两个 RelOptInfo 之间的真实关系，通过这两个函数的筛选，我们就从初步筛选的“对一个 RelOptInfo 进行初步判断”过渡到目前的“对两个 RelOptInfo 进行精确判断”。下面来看一下 have_relevant_joinclause 函数的判断依据。

- 在 RelOptInfo->joininfo 中记录了一部分连接条件（例如 a > b 这样的连接条件），如果在一个 RelOptInfo->joininfo 中的连接条件也引用了另一个 RelOptInfo，则说明这两个 RelOptInfo 是有连接条件的。
- 在等价类中不但记录了 SQL 语句中显式指定的等值约束条件，还可以基于等价类中的等价成员进行推理获得新的等值连接条件，如果两个 RelOptInfo 的 relids 在同一个等价类中，就可以说明通过这个等价类能够推理出和两个 RelOptInfo 有关的连接条件。

have_join_order_restriction 函数总是和 have_relevant_joinclause 函数成对出现的，因为在进入 have_join_order_restriction 函数的时候隐含着一个条件，那就是在两个表之间可能没有相关的连接条件，也就是说 have_relevant_joinclause 函数的返回值是 false 才会进入 have_join_order_restriction 函数中。

have_join_order_restriction 函数和 has_join_restriction 函数有点相似，都是从三个方面来判断是否有连接顺序的限制。

- 判断在两个 RelOptInfo 之间是否存在 Lateral 顺序的限制。
- 判断两个 RelOptInfo 是否包含 PlaceHolderVar 变量。
- 判断 SpecialJoinInfo 中的 min_lefthand 和 min_righthand 是否对两个 RelOptInfo（也就是 relids）有连接顺序的限制。

对 have_join_order_restriction 函数的源代码进行的分析如下：

```
bool have_join_order_restriction(PlannerInfo *root,
                         RelOptInfo *rel1, RelOptInfo *rel2)
{
   bool       result = false;
   ListCell   *l;

    //如果有 Lateral 的依赖关系，则一定有连接顺序的限制
   if (bms_overlap(rel1->relids, rel2->direct_lateral_relids) ||
      bms_overlap(rel2->relids, rel1->direct_lateral_relids))
      return true;

    //如果两个表都含有 PlaceHolderVar，则都需要在下层求值，考虑将它们先做连接
   foreach(l, root->placeholder_list)
      ......
   foreach(l, root->join_info_list)
   {
      SpecialJoinInfo *sjinfo = (SpecialJoinInfo *) lfirst(l);
      if (sjinfo->jointype == JOIN_FULL)
         continue;

      //“最小集”分别是两个表的子集，两个表需要符合连接顺序
      if (bms_is_subset(sjinfo->min_lefthand, rel1->relids) &&
         bms_is_subset(sjinfo->min_righthand, rel2->relids))
      {
         result = true;
         break;
      }
      //反过来也一样，“最小集”分别是两个表的子集，两个表需要符合连接顺序
      if (bms_is_subset(sjinfo->min_lefthand, rel2->relids) &&
         bms_is_subset(sjinfo->min_righthand, rel1->relids))
      --.....

      //如果两个表都和最小集的一端有交集，那么这两个表可能是这一端的下层的连接，
       //让它们先做连接，先把一端的最小集“凑齐”
      if (bms_overlap(sjinfo->min_righthand, rel1->relids) &&
         bms_overlap(sjinfo->min_righthand, rel2->relids))
      {
         result = true;
         break;
      }
```

```
        //反过来也一样
        if (bms_overlap(sjinfo->min_lefthand, rel1->relids) &&
            bms_overlap(sjinfo->min_lefthand, rel2->relids))
        ......
    }

    //如果在两个表上有可以产生连接关系的表并且有对应的连接关系，
    //那么先让它和可以产生连接关系的表做连接
    if (result)
    {
        if (has_legal_joinclause(root, rel1) ||
            has_legal_joinclause(root, rel2))
            result = false;
    }

    return result;
}
```

两个 RelOptInfo 无论是有连接条件或是连接顺序限制，还是没有连接条件，它们最终都要调用 make_join_rel 函数来生成连接路径，如图 8-1 所示。

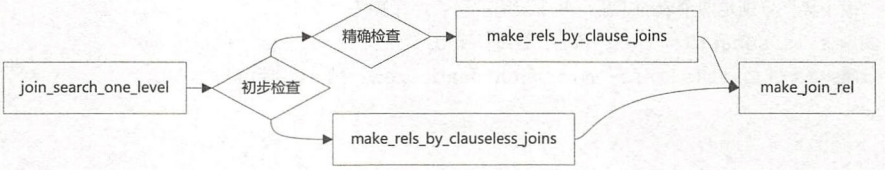


图 8-1　连接路径生成的初步和精确检查

8.1.3　“合法”连接

无论是有约束条件、连接顺序限制，还是无约束条件、无连接顺序限制，两个 RelOptInfo 的连接路径的建立最终都依赖于 make_join_rel 函数，通过 make_join_rel 函数生成它们的上一层的父 RelOptInfo，并将建立物理连接路径保存在父 RelOptInfo 中对应的成员变量中。这里需要说明一点，既然最终都会进入 make_join_rel 函数，那么何必还要在“初步检查”和“精确检查”中对有约束条件和无约束条件进行区分呢？这是因为在确定 level-1 层的表和第 1 层的所有表都无约束条件的时候，我们实际上对这种情况没有什么优化空间，只能让 level-1 层的表和第 1 层的表依次生成笛卡儿积，但是如果 level-1 层的表和第 1 层的某个表有连接条件，就可以“肯定”地排除和第 1 层没有连接条件的表，因此这里能裁减掉很多不必要的连接路径。

到目前为止，我们只有两个待选的 RelOptInfo，但还不清楚这两个 RelOptInfo 的逻辑连接关系，它可能是 Inner Join、LeftJoin、SemiJoin，亦或者这两个 RelOptInfo 之间就没有合法的逻辑连接关系，这时候就需要确定它们的逻辑连接关系，也就是做连接关系合法性的检查，这个检查的过程主要分成了两个步骤。

- 步骤 1：对 PlannerInfo->join_info_list 链表中的 SpecialJoinInfo 进行遍历，看是否能找到一个“合法”的 SpecialJoinInfo，因为除 InnerJoin 外的其他逻辑连接关系都会对应生成一个 SpecialJoinInfo，并且在 SpecialJoinInfo 中还记录了合法的连接顺序。
- 步骤 2：对 RelOptInfo 中的 Lateral 关系进行排查，查看找到的 SpecialJoinInfo 是否符合 Lateral 语义指定的连接顺序要求。

如果在 PlannerInfo->join_info_list 中找到了合法的 SpecialJoinInfo，这个 SpecialJoinInfo 也符合了 Lateral 的要求，就可以将这个 SpecialJoinInfo 作为两个 RelOptInfo 进行连接操作的依据，为其生成父连接路径，需要注意的是内连接（InnerJoin）是不记录在 PlannerInfo-> join_info_list 中的，因此在判断连接合法性的时候需要考虑到内连接的情况。

8.1.3.1　合法的连接关系

我们先来分析一下步骤 1 的实现，在 PlannerInfo->join_info_list 中目前有 4 种连接关系：LeftJoin、FullJoin、SemiJoin 和 AntiJoin，步骤 1 主要的任务是通过 SpecialJoinInfo 结构体中的最小集合（min_lefthand 和 min_righthand）判断两个 RelOptInfo 之间的关系。

```
foreach(l, root->join_info_list)
{
   SpecialJoinInfo *sjinfo = (SpecialJoinInfo *) lfirst(l);

   //两个的 RelOptInfo 的 Relids 和 min_righthand 没有交集，跳过，这是一个快速检查
   //需要注意的是如果两个 RelOptInfo 的 Relids 和 min_lefthand 没有交集，则有可能是合法的
   if (!bms_overlap(sjinfo->min_righthand, joinrelids))
      continue;

   //如果两个 RelOptInfo 的 Relids 的并集是 min_righthand 的子集，
   //例如对当前 SpecialJoinInfo 的 min_righthand = {A,B}
   //而当前的两个 RelOptInfo 分别代表的是 A 表和 B 表，那么显然这个 SpecialJoinInfo 不适用
   if (bms_is_subset(joinrelids, sjinfo->min_righthand))
      continue;

   //min_lefthand 或 min_righthand 是其中一个 RelOptInfo 的 Relids 的子集，
   //说明这个 SpecialJoinInfo 可能已经使用过了
```

```
    if (bms_is_subset(sjinfo->min_lefthand, rel1->relids) &&
        bms_is_subset(sjinfo->min_righthand, rel1->relids))
        continue;
    if (bms_is_subset(sjinfo->min_lefthand, rel2->relids) &&
        bms_is_subset(sjinfo->min_righthand, rel2->relids))
        continue;

    ……//暂时忽略 SemiJoin 的情况

    //下面这种情况是比较正常的情况，就是 min_lefthand 和 min_righthand
    //分别匹配到两个 RelOptInfo 上
    if (bms_is_subset(sjinfo->min_lefthand, rel1->relids) &&
        bms_is_subset(sjinfo->min_righthand, rel2->relids))
    {
        if (match_sjinfo)
            return false;    /* invalid join path */
        match_sjinfo = sjinfo;
        reversed = false;
    }
    //反向的情况，min_lefthand 和 min_righthand 分别匹配到两个 RelOptInfo 上
    else if (bms_is_subset(sjinfo->min_lefthand, rel2->relids) &&
             bms_is_subset(sjinfo->min_righthand, rel1->relids))
    {
        ……
        reversed = true;
    }
    ……
}
```

对于 SemiJoin 连接类型的 SpecialJoinInfo，还可以尝试对其进行唯一化处理，按照 SemiJoin 的特点，对于 LHS 中的数据，在 RHS 中找到一个符合连接条件的数据即可，不需要把 RHS 中符合连接条件的数据全部都找出来，因此可以将 RHS 中的数据“去重”，这样也不影响执行结果。

```
foreach(l, root->join_info_list)
{
    ……
    //第一轮检查：SemiJoin 的 RHS 在“唯一化”之前，不能和其他表做连接
    if (sjinfo->jointype == JOIN_SEMI)
    {
        //注意 SemiJoin 用的是 syn_righthand
```

```
        //如果 syn_righthand 是某个 RelOptInfo 的 Relids 的真子集，
        //也就是说 SemiJoin 的 syn_righthand 和其他表做了连接操作
        //间接地说明这个 SemiJoin 的 RHS 已经“唯一化”了，
        //也就是说当前的 SpecialJoinInfo 已经使用过了
        if (bms_is_subset(sjinfo->syn_righthand, rel1->relids) &&
            !bms_equal(sjinfo->syn_righthand, rel1->relids))
            continue;
        if (bms_is_subset(sjinfo->syn_righthand, rel2->relids) &&
            !bms_equal(sjinfo->syn_righthand, rel2->relids))
            continue;
    }

    //第二轮检查：SemiJoin 可能在这里就匹配上了 SpecialJoinInfo，
    //注意这里 if 条件如果能匹配上就不会进行下面的唯一化判断了
    if (bms_is_subset(sjinfo->min_lefthand, rel1->relids) &&
        bms_is_subset(sjinfo->min_righthand, rel2->relids))
    ......
    //第三轮检查
    else if (sjinfo->jointype == JOIN_SEMI &&
            //注意这里是集合相等，和上面的“真子集”对照，只有这里处理过之后
            //才会出现 SemiJoin 的 RHS 和其他变连接的情况，请注意这个先后关系
            bms_equal(sjinfo->syn_righthand, rel2->relids) &&
            //是否能够产生唯一化路径
            create_unique_path(root, rel2, rel2->cheapest_total_path,
                               sjinfo) != NULL)
    {
        ......
        unique_ified = true;
}
    else
    {
        //如果是 SemiJoin 或 AntiJoin，过了上面的 3 轮检查，这个连接关系也不适用了
        //如果是 LeftJoin，而且两个 RelOptInfo 都和 min_lefthand 没有交集，那么还可以继续检查
        if (sjinfo->jointype != JOIN_LEFT ||
         bms_overlap(joinrelids, sjinfo->min_lefthand))
            return false;    /* invalid join path */
    }
    ......
}
```

即使两个 RelOptInfo 不满足上面的检查，我们还需要做最后的“努力”，比如还需要处理

版权所有，严禁传播，违者自负法律责任！

一个特殊情况。

```
//处理 A leftjoin (B innerjoin C) left join D的情况
if (bms_overlap(rel1->relids, sjinfo->min_righthand) &&
 bms_overlap(rel2->relids, sjinfo->min_righthand))
     continue;       /* assume valid previous violation of RHS */
```

例如有这样一个 SQL 语句：SELECT * FROM TEST_A INNER JOIN TEST_B ON TRUE RIGHT JOIN TEST_C ON TRUE LEFT JOIN TEST_D ON TEST_B.a=TEST_D.a，这个语句目前只能生成一种执行计划：

```
postgres=# EXPLAIN SELECT * FROM TEST_A INNER JOIN TEST_B ON TRUE RIGHT JOIN TEST_C ON TRUE
LEFT JOIN TEST_D ON TEST_B.a=TEST_D.a;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=289.00..141029604.56 rows=6331625000 width=68)
   ->  Seq Scan on test_c  (cost=0.00..28.50 rows=1850 width=16)
   ->  Materialize  (cost=289.00..93692.81 rows=3422500 width=52)
        ->  Nested Loop  (cost=289.00..43157.31 rows=3422500 width=52)
             ->  Hash Left Join  (cost=289.00..342.94 rows=1850 width=36)
                  Hash Cond: (test_b.a = test_d.a)
                  ->  Seq Scan on test_b  (cost=0.00..28.50 rows=1850 width=16)
                  ->  Hash  (cost=164.00..164.00 rows=10000 width=20)
                       ->  Seq Scan on test_d  (cost=0.00..164.00 rows=10000 width=20)
             ->  Materialize  (cost=0.00..37.75 rows=1850 width=16)
                  ->  Seq Scan on test_a  (cost=0.00..28.50 rows=1850 width=16)
(11 rows)
```

下面尝试通过分析示例中 SQL 语句的逻辑操作结合第 4 章中给出的连接顺序交换的等式进行转换：

(A innerjoin B) rightjoin C leftjoin D
→ C leftjoin (A innerjoin B) leftjoin D
→ C leftjoin { (A innerjoin B) leftjoin D} –等式 1.3
→ C leftjoin { (B innerjoin A) leftjoin D}
→ C leftjoin { (B leftjoin D) innerjoin A} –等式 1.1

经过连接顺序交换得到的 C leftjoin { (B leftjoin D) innerjoin A}和我们在示例中得到的执行计划的连接顺序是相同的，但是如果不做特殊处理的话，这个执行计划是生成不出来的。

另外，在上述的情况之外，还需要继续做一些判断。

版权所有，严禁传播，违者自负法律责任！

- 如果 SpecialJoinInfo 结构体中的逻辑连接关系是 SemiJoin 或 AntiJoin，则这个连接关系一定是不合法的，例如对于 SQL 语句：SELECT * FROM TEST_A WHERE a > ANY (SELECT TEST_B.a FROM TEST_B LEFT JOIN TEST_C ON TEST_B.b=TEST_C.b)，它的 SemiJoin 对应的 SpecialJoinInfo 的 min_lefthand 是{TEST_A}，min_righthand 是{TEST_B, TEST_C}，在对 TEST_A 和 TEST_B 尝试做连接的“合法性”检查的时候，就会“绕”过上面的 3 轮检查，但是显然，TEST_A 和 TEST_B 还不能做 SemiJoin，所以直接放弃这个连接关系。
- 如果 SpecialJoinInfo 是 LeftJoin 并且它和 min_lefthand 有交集，那么它们的连接关系也一定是不合法的，试想 SpecialJoinInfo 中的 min_lefthand 是{TEST_A}，min_righthand 是{TEST_B, TEST_C, TEST_D}，我们要做连接的外表 RelOptInfo 代表的是{TEST_A}，内表 RelOptInfo 代表的是{TEST_C, TEST_D}，这时如果{TEST_A}和{TEST_C, TEST_D}能找到合法的连接关系，就违背了当前 min_lefthand 和 min_righthand 的“最小”这一特性，因为按照 min_lefthand 和 min_righthand 的指定，外表至少要把{TEST_B, TEST_C, TEST_D}先做连接之后才能和 TEST_A 表做连接。
- 如果 SpecialJoinInfo 是 LeftJoin 但是它和 min_righthand 没有交集，那么还可以尝试从 PlannerInfo->join_info_list 中找到其他 SpecialJoinInfo，假设 SpecialJoinInfo 中的 min_lefthand 是{TEST_A}，min_righthand 是{TEST_B, TEST_C, TEST_D}，如果当前待连接的两个 RelOptInfo 分别代表的是{TEST_B}和{TEST_C, TEST_D}，它们虽然不满足当前 SpecialJoinInfo 的 min_lefthand 和 min_righthand，但是也没有违背最小特性，因此可以去尝试到 PlannerInfo->join_info_list 中继续查找是否有其他“合法”的 SpecialJoinInfo。

8.1.3.2　合法的连接顺序

两个 RelOptInfo 有合法的连接关系只是第一步，还需要判断两个 RelOptInfo 是否有合法的连接顺序，因为类似的 Lateral 语义可以形成两个表之间的先后依赖关系。

- 两个 RelOptInfo 之间不能互相依赖，因为一旦形成“环”，这样的连接路径是无法实现的（SQL 的语法格式通常能够保证 Lateral 不会出现互相依赖的情况，在 PostgreSQL 数据库这里做一个“冗余”检查，也没有什么坏处）。
- 如果在两个 RelOptInfo 之间基于 Lateral 语义的先后依赖关系是单向的，但是我们获得的 SpecialJoinInfo 中的 min_lefthand 和 min_righthand 确定的连接顺序是和基于 Lateral 语义建立的依赖关系冲突的，则两个 RelOptInfo 不能产生合法的连接路径。
- 如果 SpecialJoinInfo 指定的连接关系是 SemiJoin，而且内表是可以“唯一化”的，那么它会使用 HashJoin 来实现它们的连接操作，而 Lateral 语义目前只能通过 NestloopJoin



严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

来实现，这时它们之间是互相冲突的，两个 RelOptInfo 也不能产生合法的连接路径。

- 如果 SpecialJoinInfo 指定的是 FullJoin，PostgreSQL 目前没有用 NestloopJoin 来实现 FullJoin，因此两个 RelOptInfo 也不能产生合法的连接路径。
- RelOptInfo 通过 direct_lateral_relids 记录它和其他 RelOptInfo 是否有直接的依赖关系，通过 lateral_relids 记录它和其他 RelOptInfo 之间的直接和间接依赖关系。两个 RelOptInfo 必须有“直接”的依赖关系才能产生连接路径，否则也无法产生合法的连接路径，例如我们已知 A 依赖 B，B 依赖 C，那么在 A 和 C 之间通过传递闭包可以推理出间接的依赖关系，但 A 和 C 不能产生连接路径，试想 A 和 C 产生了连接路径 AC，则置 B 于何地？
- 由于 Lateral 要借助 NestLoopParam 实现，而目前 NestLoopParam 只能对简单的 Var 进行传递，如果参数化路径的参数是 PlaceHolderVar，那么可能会出现“危险”的 PlaceHolderVar，这时也不能产生合法的连接路径。

另外还需要排除一种情况：如果一个 Lateral 变量引用的是外连接的 Nullable-side 的表，那么它有无法建立正常路径的风险，需要将这种风险排除，例如对于 SQL 语句：

```
SELECT * FROM TEST_A a LEFT JOIN TEST_B b ON b.c = 1, LATERAL (SELECT b.a, c.b FROM TEST_C c LIMIT 1) as cc WHERE a.a = cc.a;
```

根据谓词下推的规则，b.c=1 这个连接条件是能下推的，因此 TEST_A 和 TEST_B 之间没有连接条件，而 TEST_A 和 Subquery(TEST_C)之间有约束条件 a.a = cc.a，因此它们可能先建立连接关系，然后和 TEST_B 进行连接。

(TEST_A leftjoin TEST_B) innerjoin TEST_C
→ (TEST_A innerjoin TEST_C) leftjoin TEST_B

但是考虑到 Subquery(TEST_C)中的 Lateral 变量 b.a 导致了 TEST_B 必须在 TEST_A JOIN Subquery(TEST_C)之前出现，所以根据连接顺序交换的规则获得 TEST_B 之前的连接关系式只有下面这种情况：

TEST_B rightjoin (TEST_A innerjoin TEST_C)

而问题在于 PostgreSQL 的 RightJoin 目前无法用 NestloopJoin 的方式实现，因此这条路径是无法建立成功的。

8.2 生成新的 RelOptInfo

“合法性”的检查主要是从 PlannerInfo->join_info_list 中为两个准备连接(join)的 RelOptInfo



严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

找到一个可用的 SpecialJoinInfo，这样就能确认两个表之间的逻辑连接关系，如果通过了初步筛选、精确检查、合法性检查，就可以为这两个 RelOptInfo 建立一个 RELOPT_JOINREL 类型的父 RelOptInfo。

需要注意的是 RELOPT_JOINREL 类型的 RelOptInfo 只负责表示哪些表在这个 RelOptInfo 中进行连接操作，而不关心各个表的顺序，例如有 TEST_A LEFT JOIN TEST_B LEFT JOIN TEST_C 这样的连接关系，根据第 4 章中连接顺序交换的等式可以产生两种连接顺序。

- 连接顺序 1：TEST_A LEFT JOIN (TEST_B LEFT JOIN TEST_C)
- 连接顺序 2：(TEST_A LEFT JOIN TEST_B) LEFT JOIN TEST_C

这两个顺序都是正确的，在生成连接路径的过程中，也会基于上述的两种连接顺序生成不同的连接路径，但是这些路径都保存在同一个 RelOptInfo，因为它们都是 3 个表{TEST_A, TEST_B, TEST_C}的集合，和这 3 个表相关的连接路径都保存在 RelOptInfo->pathlist 中（也有可能是 RelOptInfo->partial_pathlist 中等）。

所有 RelOptInfo 都会保存在 PlannerInfo->join_rel_list 中，所以在每次调用 build_join_rel 函数创建新的 RelOptInfo 之前，都会遍历一下 PlannerInfo->join_rel_list 链表，如果在链表中已经存在对应的 RelOptInfo，则直接使用对应的 RelOptInfo 即可。考虑到 PlannerInfo->join_rel_list 遍历的复杂度是 O(N)，而且遍历的频率也比较高，因此 PostgreSQL 数据库对遍历的过程做了优化，如果 PlannerInfo->join_rel_list 的长度超过了 32 个，就构建一个 Hash 表（PlannerInfo->join_rel_hash）来存放这些 RelOptInfo，这样能提高查找的效率。

如果可以在 PlannerInfo->join_rel_list 或者 PlannerInfo->join_rel_hash 中找到对应的 RelOptInfo，则也并非万事大吉，因为 RelOptInfo 只关心自己包含哪些表，但是不关心它们的连接顺序，因此还需要单独处理约束条件。例如对于 SQL 语句 SELECT * FROM TEST_A a, TEST_B b, TEST_C c, TEST_D d WHERE a.a > b.a AND b.b > c.b AND c.c >d.d，在生成连接路径的过程中会生成由{TEST_A, TEST_B, TEST_C}组成的一个 RelOptInfo，在这个 RelOptInfo 中包含不同连接顺序的各种路径，由于连接顺序的不同，约束条件的位置是不同的，如图 8-2 所示可以看出两个约束条件 a.a > b.a 和 b.b > c.b 的位置是不同的，所以，不同于基表（RELOPT_BASEREL，执行计划树的叶子节点，或者说是扫描节点）将约束条件保存在 RelOptInfo->baserestrictinfo 中，连接路径的约束条件不是保存在 RelOptInfo 中的，它们是保存在每一条连接路径中的，连接路径的结构体是 JoinPath，因此连接路径的约束条件就是保存在 JoinPath->joinrestrictinfo 中的。

另外需要注意的是，在 SQL 语句中还有一个约束条件是 c.c >d.d，这个约束条件对于 RelOptInfo 中的所有路径而言都是相同的，因此它可以保存到 RelOptInfo->joininfo 中。对于代表{TEST_A,



严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

TEST_B, TEST_C}的 RelOptInfo 而言，它还无法应用 c.c > d.c 这样的约束条件，还需要等待包含 TEST_D 的 RelOptInfo 出现。

在 build_join_rel 函数中，分别调用 build_joinrel_restrictlist 函数和 build_joinrel_joinlist 函数来对这些约束条件进行筛选。

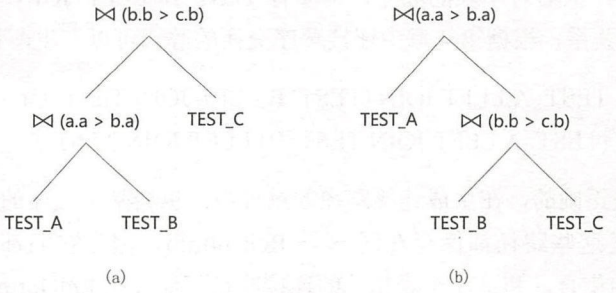


图 8-2　连接条件的应用

如果无法在 PlannerInfo->join_rel_list 或者 PlannerInfo->join_rel_hash 中找到对应的 RelOptInfo，则需要创建一个新的 RelOptInfo。创建一个 RELOPT_JOINREL 类型的 RelOptInfo 的过程就是从连接的内表和外表对应的 RelOptInfo 中提取信息的过程，这个过程较为简单，其中需要注意对 PlaceHolderVar 的处理，它不是把 PlaceHolderVar 像普通的表达式一样增加在 RelOptInfo->reltarget 中（build_joinrel_tlist 函数），而是通过 add_placeholders_to_joinrel 函数判断 PlaceHolderVar 是否有必要加入到当前的 RelOptInfo->reltarget 中。

需要注意的是，在 add_placeholders_to_joinrel 函数中对 PlaceHolderVar 能不能加入进行了两个判断，分别是 bms_nonempty_difference 函数判断和 bms_is_subset 函数判断，这是因为：假如有 TEST_A、TEST_B、TEST_C 三个表做连接，TEST_C 上有 a 属性需要使用 PlaceHolderVar，那么在为{TEST_A, TEST_B}对应的 RelOptInfo 生成 reltarget 的时候，是可以通过 bms_nonempty_difference 的检查的，但实际上{TEST_A, TEST_B}对应的 RelOptInfo 的 reltarget 是不需要这个 PlaceHolderVar 的，因此再使用 bms_is_subset 做一下检查。

```
//这里分别使用了 bms_nonempty_difference 和 bms_is_subset 进行判断
//PlaceHolderVar 是否有必要加入到 RelOptInfo->reltarget
if (bms_nonempty_difference(phinfo->ph_needed, relids))
{
   //bms_nonempty_difference 筛选得可能不够准确
   if (bms_is_subset(phinfo->ph_eval_at, relids))
      ......
}
```




严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

8.3　虚表

准备好了 RelOptInfo 和合法的 SpecialJoinInfo，就可以向准备好的 RelOptInfo 中增加物理连接路径了，路径的生成在 populate_joinrel_with_paths 函数中实现，它根据 SpecialJoinInfo 中指定的逻辑连接类型调用 add_paths_to_joinrel 函数来添加物理连接路径。

但是在添加路径之前，还需要判断一下在两个 RelOptInfo 中是否有执行结果一定为空的表，例如一个 RelOptInfo 的约束条件为常量 false，它最终的执行结果一定是空集，我们可以称这种类型的 RelOptInfo 为“虚表”，对于不同的连接类型，虚表和其他 RelOptInfo 的连接结果可能还是一个虚表，如表 8-1 所示。

表 8- 1　虚表和连接类型的关系

连接类型	优化方式
Inner Join	内连接是最常见的方式，如果在内表、外表或者约束条件中有常量false，则连接结果就是虚表。 例句：SELECT * FROM (SELECT * FROM TEST_A WHERE 2>3) ta INNER JOIN TEST_B ON TRUE;
Left Join	如果左连接的外表是虚表或者有常量false的约束条件，整个连接产生的连接结果一定为虚表。 如果左连接的内表是虚表或者有常量false的约束条件，则可以将内表设置为虚表。 例句：SELECT * FROM (SELECT * FROM TEST_A WHERE 2>3) ta LEFT JOIN TEST_B ON TRUE;
Full Join	如果全连接的内表和外表全部是虚表，则连接结果为虚表； 如果全连接有常量false的约束条件，则连接结果为虚表。 例句：SELECT * FROM (SELECT * FROM TEST_A WHERE 2>3) ta LEFT JOIN (SELECT * FROM TEST_B WHERE 3 >4) tb ON TRUE;
Semi Join	和内连接相似。 例句：SELECT * FROM TEST_A WHERE (a,0) IN (SELECT a,2 FROM TEST_B);
Anti Join	和左连接相似。 例句：SELECT * FROM TEST_A WHERE NOT EXISTS (SELECT a FROM TEST_B WHERE TEST_A.a > TEST_B.a AND 2 > 3);

需要注意的是 restriction_is_constant_false 函数的第 2 个参数 only_pushed_down 代表针对哪些约束条件进行检查，在外连接的情况下，only_pushed_down = true 值代表只针对能够下推的过滤条件（可以参考第 4 章约束条件下推的内容）做检查，这样做是因为即使在连接条件是常



非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

量 false 的情况下，连接的结果可能并不是一个虚表，因为在外连接中，对于不满足连接条件的 Nonnullable-side 的元组仍然会被显示出来，并且会在 Nullable-side 中补充 NULL 值，因此在这里如果要判断两个 RelOptInfo 连接的结果是虚表，则需要对可以下推的约束条件也就是过滤条件进行判断，例如：

```
postgres=# EXPLAIN SELECT * FROM TEST_A LEFT JOIN TEST_B ON FALSE;
                        QUERY PLAN
------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..47.00 rows=1850 width=32)
   Join Filter: false
   ->  Seq Scan on test_a  (cost=0.00..28.50 rows=1850 width=16)
   ->  Result  (cost=0.00..0.00 rows=0 width=16)
         One-Time Filter: false
(5 rows)
```

从示例中还可以看出，虽然连接条件没有对 TEST_A LEFT JOIN TEST_B 起到作用，但是对 Nullable-side 的 TEST_B 起到作用，对 TEST_B 表的扫描变成了虚表（虚表路径在生成执行计划的阶段会建立基于 One-Time Filter 的 Result 节点），从外连接的语义也可以看出，在 ON FALSE 的时候，也就是说 Nonnullable-side 没有和 Nullable-side 匹配的元组，这和 Nullable-side 是虚表的效果是一样的。同理，无论是过滤条件还是连接条件，只要有常量 false 约束条件，那么外连接 Nullable-side 的表就可以修正为虚表，这里需要说明的是这里的 Nullable-side 的判断使用了 SpecialJoinInfo->syn_righthand 而不是 SpecialJoinInfo->min_righthand，因为我们记录 min_righthand 的目的是保证连接顺序的正确性，而 syn_righthand 的目的是记录语句中原始的逻辑连接关系，只要有约束条件是常量 false，那么对一个连接操作而言，它会作用到整个连接上，而不一定是连接的“最小集（min_righthand）”上。

在 add_paths_to_joinrel 函数中，Inner Join、Left Join、Full Join 分别尝试了交换两个 RelOptInfo 的位置来尝试更多的可能路径，对于 Inner Join 和 Full Join 而言，在交换 RelOptInfo 的位置之后它们仍然是 Inner Join 和 Full Join 的关系，但是在 Left Join 的两个 RelOptInfo 交换位置之后就变成了 Right Join。在逻辑优化阶段做外连接消除时，我们将所有的 Right Join 通过交换连接表位置的方式都转换成了 Left Join，因此在之后很长一段时间内我们只处理 Left Join，而无须考虑 Right Join，但到了 populate_joinrel_with_paths 函数之后，Right Join 又“回来了”。

8.4 Semi Join 和唯一化路径

在介绍连接路径的建立过程之前，我们先谈一谈唯一化路径（UniquePath），在合法性检



非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

查（join_is_legal 函数）的过程中，我们就检查了 Semi Join 的一个优化措施，就是将 Semi Join 的内表唯一化，因为 Semi Join 的语义是“找到一个符合连接条件即可”，因此这个条件是能唯一化的，唯一化的过程会增加查询的代价，但是带来的好处是 Semi Join 可以被消除掉，也就是说 Semi Join 的内表如果是唯一化的，那么它和内连接的结果实际上是等价的，转换成内连接之后，连接顺序的交换会更自由，连接路径的建立就获得了更大的发挥空间，因此路径的唯一化实际上是尝试做一个收支的考量——是由唯一化而增加的代价多还是由于转换成为内连接而带来的好处多。

在 join_is_legal 函数和 populate_joinrel_with_paths 函数中都对 Semi Join 的内表做了检查（在 join_is_legal 函数检查时产生的唯一化路径会被保存起来，populate_joinrel_with_paths 函数可能会直接利用而不是自己再去重复创建），如果 Semi Join 的内表满足以下条件，那么它已经隐含了唯一性的特征，我们可以直接认为它就是唯一化路径。

- 如果 Semi Join 的内表是普通表，且表上有 Unique 索引，而且约束条件能和索引完全匹配，那么它本身就是唯一化路径。PostgreSQL 数据库通过 relation_has_unique_index_for 函数来判断约束条件是否能和 Unique 索引匹配。
  - 必须是唯一索引（判断 Unique 标记）。
  - 必须保证 IndexOptInfo->immediate == true，因为如果 IndexOptInfo->immediate == false，那么就代表这个唯一性约束是可延迟的，这种情况可能产生暂时的不唯一。
  - 如果是带有谓词的局部索引，那么谓词必须符合约束条件的要求（这是索引在这个查询中能够被利用的最低要求）。
  - 索引的键要么和约束条件（restrictlist 变量）能够匹配，要么和 Semi Join 的内表的相关的表达式（SpecialJoinInfo->semi_rhs_exprs）能够匹配。
- 如果 Semi Join 的内表是子查询，子查询中有具有“唯一”性质的子句，且该子句和内表的约束条件能够匹配，就可以考虑唯一化路径。
  - 如果子查询带有 distinctClause 子句，那么它的结果具有唯一性质。
  - 如果子查询带有 groupClause 子句（不含有 Grouping sets 子句），那么进行分组的属性具有唯一性。
  - 如果在查询中有 Grouping sets，对其进行唯一化分析会比较复杂，因此 PostgreSQL 数据库目前没有分析这种情况，但处理了 Grouping Sets 为空且只有 1 个的情况，在这种情况下子查询只会产生一条结果，可以认为是“唯一化”的。
  - 如果在查询中没有分组子句（即没有 groupClause 和 Grouping sets），那么普通的聚集函数和 Having 子句保证最终结果只有一行，可以认为是“唯一化”的。



非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

- UNION, INTERSECT, EXCEPT 这些集合操作如果不带有“ALL”关键字，那么它们的结果具有唯一性。

如果 Semi Join 的内表不具有隐式的唯一化特征，那么我们可以显式地给它加上唯一化的方法，构建唯一化路径。在 PostgreSQL 数据库中数据的唯一化通常有两种方法：Sort 和 Hash，因此要构建显式的唯一化路径，需要满足 sjinfo->semi_can_btree == true 或 sjinfo->semi_can_hash == true，满足 sjinfo->semi_can_btree 则可以尝试用 Sort 的方法去重，满足 sjinfo->semi_can_hash 则可以尝试用 Hash 的方法去重，无论是使用 Sort 的方法还是使用 Hash 的方法都会增加代价，因此还需要注意对这部分代价进行估算。

在 add_paths_to_joinrel 中如果发现 SemiJoin 的内表是唯一化的，Semi Join 在内表“找到一个符合连接条件即可”的这个特性就成为必然，因为此时内表也只有一条元组供它匹配，这时就可以将 Semi Join 变为 Inner Join，如下面的示例所示：

```
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE EXISTS (SELECT a FROM TEST_B WHERE TEST_A.a
= TEST_B.a);
                          QUERY PLAN
------------------------------------------------------------------------------
 Hash Join  (cost=37.62..81.27 rows=925 width=16)
   Hash Cond: (test_a.a = test_b.a)
   ->  Seq Scan on test_a  (cost=0.00..28.50 rows=1850 width=16)
   ->  Hash  (cost=35.12..35.12 rows=200 width=4)
         ->  HashAggregate  (cost=33.12..35.12 rows=200 width=4)
               Group Key: test_b.a
               ->  Seq Scan on test_b  (cost=0.00..28.50 rows=1850 width=4)
(7 rows)
```

从该示例的执行计划可以看出，在执行计划中已经没有了 Semi Join 的逻辑了，主要是因为内表做了基于 Hash 聚集的唯一化处理，实际上在逻辑优化的阶段，我们通过 reduce_unique_semijoins 函数做过类似的处理，具体请参考第 4 章中 Semi Join 消除部分的内容。

在 add_paths_to_joinrel 函数中还调用 compute_semi_anti_join_factors 函数对 Semi Join、Anti Join、带有唯一化路径的连接操作做了处理，因为它们都符合“找到一个符合连接条件即可”。由于 Semi Join、Anti Join 和内表唯一化的连接操作这 3 种情况是通过 NestloopJoin 或者 HashJoin 实现的，它们的代价计算和普通的 Nestloop Join 或者普通的 Hash Join 不同，因此这时可以先计算出一些代价因子，到代价计算的时候可以利用这些代价因子单独计算这 3 种情况的代价。

这些代价因子的计算是在 compute_semi_anti_join_factors 函数中实现的，通过 clauselist_selectivity 函数分别获得了两个 RelOptInfo 在 Inner Join 下的选择率 nselec 和在 Semi Join 下的选



非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

择率 jselec，虽然约束条件相同，但是 nselec 和 jselec 是不同的，因为在 Inner Join 的情况下选择率的基数是两个 RelOptInfo 的笛卡儿积的结果数，而在 Semi Join 的情况下选择率的基数是外表的元组数。然后我们根据 nselect 和 jselect 来计算一下外表的一条元组平均能够匹配内表的几条元组，假设外表的元组数是 m，内表的元组数是 n，两个表做 Inner Join 并应用约束条件获得的结果集数量是 X，而做 Semi Join 并应用约束条件获得的结果集数量是 Y，那么因为 X 是把内表所有的匹配都计算在内的，而 Y 是在内表匹配一条即可的，因此在平均的情况下每个外表元组可以匹配 X/Y 条元组，也就是说：

$$avgmatch = \frac{X}{Y} = \frac{\frac{X}{m \times n}}{\frac{Y}{m}} \times n = (\frac{nselec}{jselec}) \times n$$

这样在计算连接路径代价的时候，如果连接的类型是 Semi Join，就能使用这个 avgmatch 来估计外表的一条元组需要在内表查找几条记录才能找到第一条符合约束条件的元组（avgmatch 在 Nestloop Join 和 Hash Join 的代价计算的时候会用到，因此它在开始创建路径之前就把这个值计算并保存下来，这样在不同的物理路径进行代价计算的时候都能复用这个值）。

8.5　建立连接路径

add_paths_to_joinrel 函数负责建立物理连接路径，虽然物理连接路径只有 Nestloop Join、Merge Join、Hash Join 这几种方式，但是还需要同时考虑并行路径、参数化路径等，因此需要考虑的内容还是非常多的，连接路径建立的函数关系如图 8-3 所示，它主要是借由 sort_inner_and_ outer 函数、match_unsorted_outer 函数和 hash_inner_and_outer 函数来实现的。

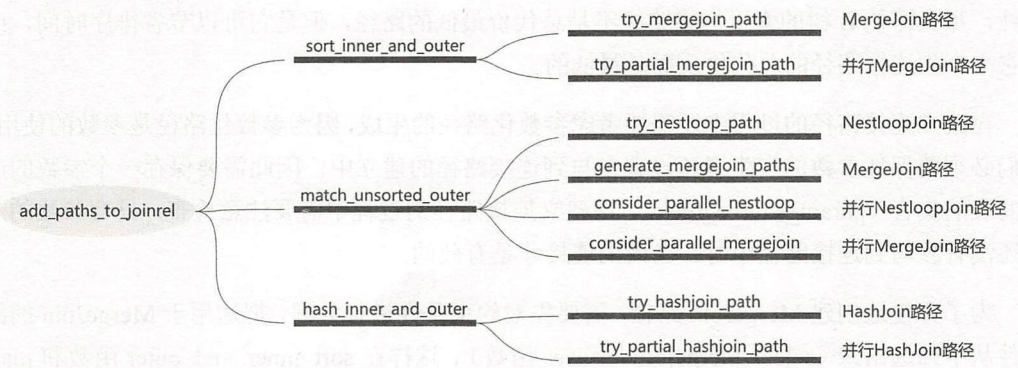


图 8-3　连接路径建立的函数调用关系

非卖品，仅供非商业用途或交流学习使用

建立连接路径是根据在待选的两个子 RelOptInfo 中的路径生成新的父 RelOptInfo 的路径的过程，我们知道在创建扫描路径的时候，在每个 RelOptInfo 中可能有多个扫描路径，包括并行路径、参数化路径以及普通的扫描路径等，那么应该选择哪个路径来生成新的父 RelOptInfo 的路径呢？如图 8-4 所示。

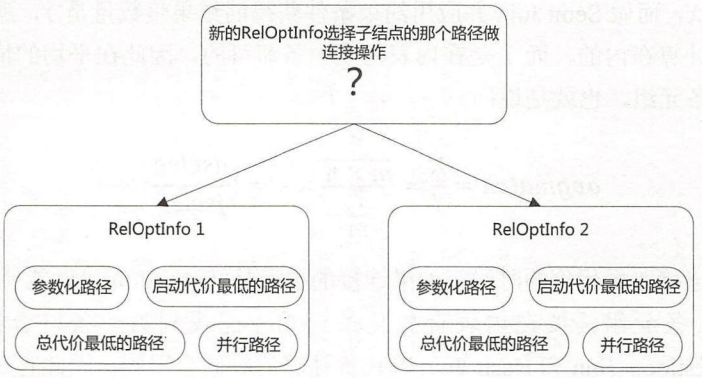


图 8-4　扫描路径的选择

例如 sort_inner_and_outer 函数主要生成 MergeJoin 路径，这个函数的特点是必须显式地对子 RelOptInfo 进行排序，因此这时候只需要选择子 RelOptInfo 中的总代价最低的路径（RelOptInfo->cheapest_total_path）即可。

又如 match_unsorted_outer 函数也尝试生成 MergeJoin 路径，但是这时它只显式地对内表排序，也就是说只有在外表的 PathKeys 能够匹配连接条件的情况下（PathKeys 代表该路径的结果是有序的），只需要对内表显式地排序就能获得 MergeJoin 路径，这时内表的路径就不一定是总代价最低的路径，比如内表的路径是 B 树索引扫描路径，而且它的 PathKeys 满足当前的连接条件，那么即使 B 树的索引扫描路径不是总代价最低的路径，但是它可以节省排序时间，或许用它来产生连接路径的总代价反而是最低的。

在建立连接路径的过程中还需要考虑参数化路径的生成，因为参数化路径是参数的使用者，我们必须要保证参数的产生者还没有参与到连接路径的建立中，因此需要保存一个参数的产生者的表的集合（param_source_rels），在建立连接路径的过程中需要注意检查，只有参数的产生者还没有参与到连接路径中时，当前的连接才是有效的。

为了方便地创建 MergeJoin 路径，需要先对约束条件进行处理，把适用于 MergeJoin 的约束条件从中筛选出来（select_mergejoin_clauses 函数），这样在 sort_inner_and_outer 函数和 match_unsorted_outer 函数中都可以利用这个 Mergejoinable 连接条件。

非卖品，仅供非商业用途或交流学习使用

```
foreach(l, restrictlist)
{
   RestrictInfo *restrictinfo = (RestrictInfo *) lfirst(l);

   //is_pushed_down 是区分连接条件和过滤条件的标志，如果当前是外连接，那么跳过过滤条件
   if (isouterjoin && restrictinfo->is_pushed_down)
      continue;

   //can_join 是在 distribute_qual_to_rels 函数->make_restrictinfo 函数
   //    ->make_restrictinfo_internal 函数中做的一个初步的判断
   //mergeopfamilies 是在 distribute_qual_to_rels 函数->check_mergejoinable 函数中做的判断
   if (!restrictinfo->can_join ||
      restrictinfo->mergeopfamilies == NIL)
   {
      //如果是 FULL JOIN ON FALSE 这种情况，则不设置 have_nonmergeable_joinclause 为 true
      if (!restrictinfo->clause || !IsA(restrictinfo->clause, Const))
         have_nonmergeable_joinclause = true;
      continue;          /* not mergejoinable */
   }

   //约束条件是 inner op outer 或者 outer op inner 的形式
   if (!clause_sides_match_join(restrictinfo, outerrel, innerrel))
   {
      have_nonmergeable_joinclause = true;
      continue;          /* no good for these input relations */
   }

   //使用最终的等价类，例如在创建索引路径的时候，生成的 PathKeys 都是基于最终等价类的
   //“规范化”PathKeys，这样约束条件就能和 PathKeys 进行匹配了
   update_mergeclause_eclasses(root, restrictinfo);
   if (EC_MUST_BE_REDUNDANT(restrictinfo->left_ec) ||
      EC_MUST_BE_REDUNDANT(restrictinfo->right_ec))
   {
      have_nonmergeable_joinclause = true;
      continue;          /* can't handle redundant eclasses */
   }

   result_list = lappend(result_list, restrictinfo);
}
```


非卖品，仅供非商业用途或交流学习使用

8.5.1 sort_inner_and_outer 函数

sort_inner_and_outer 函数主要是生成 MergeJoin 路径，它显式地对两个子 RelOptInfo 进行排序，因此只考虑子 RelOptInfo 中的 cheapest_total_path 路径即可，需要注意的是，并行的 MergeJoin 路径的生成依赖于路径的外表（LHS 端）的 RelOptInfo 中是否有并行路径，因此在建立并行的 MergeJoin 时选择的是子 RelOptInfo 中代价最低的并行路径。

```
if (joinrel->consider_parallel && //是否可以考虑并行路径
   //外表是 UniquePath，不能划分到多个线程里
   save_jointype != JOIN_UNIQUE_OUTER &&
   //FullJoin 和 RightJoin 不行，因为会在外表上产生 NULL 值扩展
   save_jointype != JOIN_FULL &&
   save_jointype != JOIN_RIGHT &&
   //在 LHS 的 RelOptInfo 中包含并行路径
   outerrel->partial_pathlist != NIL &&
   //不包含 Lateral 参数
   bms_is_empty(joinrel->lateral_relids))
{
   //注意 RelOptInfo->partial_pathlist 中的路径是按代价高低排序的
   //选取代价最低的并行路径，用于建立并行 MergeJoin 路径
   cheapest_partial_outer = (Path *) linitial(outerrel->partial_pathlist);

   //选择 RHS 的 RelOptInfo 中代价最低的“并行安全”路径
   if (inner_path->parallel_safe)
      cheapest_safe_inner = inner_path;
   else if (save_jointype != JOIN_UNIQUE_INNER)
      cheapest_safe_inner =
         get_cheapest_parallel_safe_total_inner(innerrel->pathlist);
}
```

MergeJoin 连接既然包含了排序的操作，那么是否能够尝试为它的连接结果生成一个 PathKeys 呢？根据 MergeJoin 连接的执行特点，它的连接结果的顺序和外表的顺序一致（排除 Right Join 和 Full Join，因为这两个连接操作都可能要在外表也就是 LHS 端补 NULL 值），因此这个 PathKeys 是可以建立的，下面通过一个左连接的示例来展示一下。

```
postgres=# SELECT * FROM TEST_A;
 a | b | c | d
---+---+---+---
 4 | 3 | 2 | 1
 3 | 3 | 2 | 1
 2 | 3 | 2 | 1
```


非卖品，仅供非商业用途或交流学习使用

```
 1 | 3 | 2 | 1
(4 rows)

postgres=# SELECT * FROM TEST_B;
 a | b | c | d
---+---+---+---
 3 | 4 | 5 | 6
 5 | 6 | 7 | 8
 7 | 8 | 9 | 0
(3 rows)

postgres=# EXPLAIN SELECT * FROM TEST_A LEFT JOIN TEST_B ON TEST_A.a = TEST_B.a;
                        QUERY PLAN
------------------------------------------------------------------------
 Merge Left Join  (cost=2.13..2.19 rows=4 width=32)
   Merge Cond: (test_a.a = test_b.a)
   ->  Sort  (cost=1.08..1.09 rows=4 width=16)
         Sort Key: test_a.a
         ->  Seq Scan on test_a  (cost=0.00..1.04 rows=4 width=16)
   ->  Sort  (cost=1.05..1.06 rows=3 width=16)
         Sort Key: test_b.a
         ->  Seq Scan on test_b  (cost=0.00..1.03 rows=3 width=16)
(8 rows)
--连接结果的顺序是对 TEST_A 排序之后的顺序，也就是符合外表的顺序
postgres=# SELECT * FROM TEST_A LEFT JOIN TEST_B ON TEST_C.a = TEST_B.a;
 a | b | c | d | a | b | c | d
---+---+---+---+---+---+---+---
 1 | 3 | 2 | 1 |   |   |   |
 2 | 3 | 2 | 1 |   |   |   |
 3 | 3 | 2 | 1 | 3 | 4 | 5 | 6
 4 | 3 | 2 | 1 |   |   |   |
(4 rows)
```

在 add_paths_to_joinrel 函数中我们已经通过 select_mergejoin_clauses 函数获得了适用于 MergeJoin 的约束条件，结合 MergeJoin 产生的 PathKeys 的特点，约束条件的应用顺序就变得尤为重要。假设有两个子约束条件 A 和 B 是 MergeJoinable 的约束条件，那么在生成路径的过程中可以用{A, B}来生成一条 MergeJoin 路径，也可以用{B , A}来生成一条 MergeJoin 路径，这两个路径的估计代价是相同的，但这两条路径产生的连接结果的排序方式是不同的，也就是说两条路径产生的 PathKeys 是不同的，在前面的章节中已经介绍过 PathKeys 起到的是“有序”的提示作用，因此不同的 PathKeys 对外层的操作会产生不同的影响，例如：

非卖品，仅供非商业用途或交流学习使用

```
postgres=# EXPLAIN SELECT * FROM TEST_A LEFT JOIN TEST_B ON TEST_A.a = TEST_B.a AND TEST_A.b
= TEST_B.b FULL JOIN TEST_C ON TEST_A.b = TEST_C.b;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Merge Full Join  (cost=55874.19..120329.01 rows=4075143 width=48)
   Merge Cond: (test_c.b = test_a.b)
   ->  Sort  (cost=128.89..133.52 rows=1850 width=16)
         Sort Key: test_c.b
         ->  Seq Scan on test_c  (cost=0.00..28.50 rows=1850 width=16)
   ->  Materialize  (cost=55745.30..60165.12 rows=440556 width=32)
         ->  Merge Left Join  (cost=55745.30..59063.73 rows=440556 width=32)
               Merge Cond: ((test_a.b = test_b.b) AND (test_a.a = test_b.a))
               ->  Sort  (cost=55616.41..56717.80 rows=440556 width=16)
                     Sort Key: test_a.b, test_a.a
                     ->  Seq Scan on test_a  (cost=0.00..6784.56 rows=440556 width=16)
               ->  Sort  (cost=128.89..133.52 rows=1850 width=16)
                     Sort Key: test_b.b, test_b.a
                     ->  Seq Scan on test_b  (cost=0.00..28.50 rows=1850 width=16)
(14 rows)
```

示例 SQL 语句中的 TEST_A LEFT JOIN TEST_B ON TEST_A.a = TEST_B.a AND TEST_A.b = TEST_B.b 这部分如果要生成 MergeJoin 的路径，则它的外表 TEST_A 可以选择的排序顺序可以是{TEST_A.a, TEST_A.b}和{TEST_A.b, TEST_A.a}这两种 PathKeys，通过观察执行计划我们还发现，TEST_A LEFT JOIN TEST_B 的连接结果还需要和 TEST_C 做 FullJoin，在 FullJoin 的约束条件中包含了 TEST_A.b，因此如果我们在 TEST_A LEFT JOIN TEST_B 的连接时选择{TEST_A.b, TEST_A.a}作为 MergeJoin 的 PathKeys，MergeJoin 的连接结果的第一排序键就是 TEST_A.b，那么在和上层的 TEST_C 表做 MergeJoin 连接的时候就能省略对 TEST_A.b 的排序，就能减少连接的代价。

8.5.1.1 外表的初始 PathKeys

既然根据 Mergejoinable 连接条件能生成不同顺序的 PathKeys，而且 MergeJoin 的连接结果的顺序和外表的顺序一致，那么我们可以将 Mergejoinable 连接条件中外表的等价类提取出来（RestrictInfo 中记录了外表和内表的等价类），形成一个初始的 PathKeys，然后使用这个初始的 PathKeys 衍生出其他顺序的 PathKeys，再用衍生出的 PathKeys 对 MergeJoinable 的约束条件进行排序，最后根据排序后的 MergeJoinable 约束条件生成内表的 PathKeys。

我们这里使用 PathKeys 来衍生出不同顺序的约束条件，而不是直接使用约束条件自身来衍生的原因是：一方面 PathKeys 更容易和上层的 PlannerInfo->query_pathkeys 进行比较，因为这

时的等价类都是“最终”等价类，形成的 PathKeys 都是“规范”的 PathKeys；另一方面约束条件中的有些排序结果是冗余的，例如对于 A.a = B.b AND A.a = B.c AND A.a = C.d AND A.b = D.e 这样的约束条件，它的外表的 PathKeys 只可能是{A.a, A.b}或者{A.b, A.a}，但是我们通过约束条件排序可能得到很多种冗余的路径。

由 MergeJoinable 连接条件向 PathKeys 转换的过程是在 select_outer_pathkeys_for_merge 函数中实现的，这个函数主要分成了 3 个步骤。

步骤 1，根据 MergeJoinable 连接条件生成一个等价类数组 ecs，并且为每个等价类打分。在约束条件（RestrictInfo）中记录了外表的属性是处于约束条件的 LHS 还是 RHS，可以根据这些信息获得约束条件中外表的属性对应的等价类，然后对这个等价类进行打分，打分越高说明这个等价类越重要，打分的标准是：这个等价类的成员和哪些 RelOptInfo 相关（减去当前待连接的两个 RelOptInfo），每和 1 个 RelOptInfo 相关就增加 1 分，例如对于 SELECT * FROM TEST_A a, TEST_B b, TEST_C c WHERE a.a = b.a AND a.b = b.b AND a.a = c.a，这个语句会形成{TEST_A.a, TEST_B.a, TEST_C.a}和{TEST_A.b, TEST_B.b}这样的两个等价类，假如对 TEST_A 和 TEST_B 做 MergeJoin 时，由于在等价类{TEST_A.a, TEST_B.a, TEST_C.a}中除了待连接的 TEST_A 的属性和 TEST_B 的属性，还包含一个 TEST_C 中的属性，因此这个等价类的重要性就是 1 分，另一个等价类{TEST_A.b, TEST_B.b}则是 0 分。

步骤 2，用生成的等价类数组 ecs 去匹配 PlannerInfo->quer_pathkeys（PlannerInfo->quer_pathkeys 是 Non-SPJ 优化阶段期望的排序顺序，它提示在 SPJ 优化阶段“多”考虑一下它的期望，MergeJoin 属于 SPJ 优化阶段的物理连接路径，这时它需要迎合 Non-SPJ 优化阶段的期望）。查看形成的等价类数组是否“全面”包含了 PlannerInfo->query_pathkeys 中对应的所有等价类，如果 PlannerInfo->query_pathkeys 有一个 PathKey 没有在数组 ecs 中找到，那么匹配失败，在第 3 个步骤中就完全依赖于等价类打分的顺序来生成初始 PahtKeys。但是如果 ecs 和 PlannerInfo->query_pathkeys 匹配成功，那么通过 MergeJoinable 连接条件生成的初始的 PathKeys 一定要以 PlannerInfo->query_pathkeys 为基础，最终的结果形成一个以 PlannerInfo-> query_pathkeys 为开头的结果，这时就需要将 PlannerInfo->query_pathkeys 中的等价类对应的打分置为-1，因为这些等价类已经保存在 PathKeys 的开始的位置，第 3 个步骤无须再考虑这些等价类，以 SELECT * FROM TEST_A a, TEST_B b, TEST_C c WHERE a.a = b.a AND a.b = b.b AND a.c = b.c AND a.a = c.d 为例，这个语句会形成{TEST_A.a, TEST_B.a, TEST_C.d}、{TEST_A.b, TEST_B.b}、{TEST_A.c, TEST_B.c}这样的 3 个等价类，它们的得分分别是 1、0、0，如果 Non-SPJ 阶段没有给出 Planner->query_pathkeys 的“期望”，那么它们按照得分来排序，假如 Non-SPJ 给出了期望是以 TEST_B.b 有序，那么{TEST_A.b, TEST_B.b}这个等价类现在就固定排到前边，作为

第一排序键，这样就能迎合 Non-SPJ 阶段的需要。

步骤 3，在所有打分的等价类中找到分数最高的等价类，加入到衍生的 PathKeys 中，并将这个等价类的分数置为-1，然后重新查找分数最高的等价类，依此类推，直至所有等价类的分数都小于 0（也就是说所有等价类都处理完成），最终返回 PathKeys 作为初始的 PathKeys。

需要注意的是在步骤 2 中虽然考虑了 Non-SPJ 的“期望”，但在后面还是会调整 PathKeys 的顺序，用来产生更多的可能性，也就是说如果在 Non-SPJ 阶段有期望，就迎合 Non-SPJ 的期望，但随后我们还需要考虑更多的其他情况。

8.5.1.2 不同顺序的约束条件

在获得初始的 PathKeys 之后可以通过交换 PathKeys 的顺序来形成不同顺序的 MergeJoinable 的连接条件。但是如果 PathKeys 比较多，我们穷举所有顺序的算法复杂度就会比较高，为了减少生成执行计划的时间（降低生成执行计划的时间也很重要，否则可能出现面粉贵过面包的情况），这里我们需要采用一些启发式的规则，就是所有的 PathKeys 中的 PathKey 都有机会作为 PathKeys 中的第一个 PathKey 出现，但是后面的 PathKey 的顺序是随意的。例如我们有一个约束条件如下：

A.a = B.a AND A.b = B.b AND A.c = B.c AND A.d = B.d

这个约束条件对应的等价类有{A.a, B.a}、{A.b, B.b}、{A.c, B.c}、{A.d, B.d}共 4 个，外表 A 对应的 PathKeys 的内容是{A.a} -> {A.b} -> {A.c} -> {A.d}，那么我们只需要考虑如下 4 种 PathKeys 顺序：

{A.a} -> {A.b} -> {A.c} -> {A.d} 对应 A.a = B.a AND A.b = B.b AND A.c = B.c AND A.d = B.d
{A.b} -> {A.a} -> {A.c} -> {A.d} 对应 A.b = B.b AND A.a = B.a AND A.c = B.c AND A.d = B.d
{A.c} -> {A.a} -> {A.b} -> {A.d} 对应 A.c = B.c AND A.a = B.a AND A.b = B.b AND A.d = B.d
{A.d} -> {A.a} -> {A.b} -> {A.c} 对应 A.d = B.d AND A.a = B.a AND A.b = B.b AND A.c = B.c

```
//从中提取每一个 PathKey 到最前面，如果本身就是第一个，就先用这一个
if (l != list_head(all_pathkeys))
   outerkeys = lcons(front_pathkey,
                     list_delete_ptr(list_copy(all_pathkeys),
                                     front_pathkey));
else
   outerkeys = all_pathkeys;   /* no work at first one... */

//根据重新排序的 outerkeys，产生对应顺序的约束条件
```

```
cur_mergeclauses = find_mergeclauses_for_pathkeys(root,
                                                  outerkeys,
                                                  true,
                                                  extra->mergeclause_list);
```

在产生了候选的某个顺序的约束条件之后，我们还可以生成内表的 PathKeys 用来匹配当前内表是否已经是有序的。

A.a = B.a AND A.b = B.b AND A.c = B.c AND A.d = B.d 对应{B.a} -> {B.b} -> {B.c} -> {B.d}
A.b = B.b AND A.a = B.a AND A.c = B.c AND A.d = B.d 对应{B.b} -> {B.a} -> {B.c} -> {B.d}
A.c = B.c AND A.a = B.a AND A.b = B.b AND A.d = B.d 对应{B.c} -> {B.a} -> {B.b} -> {B.d}
A.d = B.d AND A.a = B.a AND A.b = B.b AND A.c = B.c 对应{B.d} -> {B.a} -> {B.b} -> {B.c}

```
//根据当前约束条件的顺序产生内表的 PathKeys
//这个 PathKeys 对 MergeJoin 的连接结果没有提示作用，
//但是它对待连接的内表 RelOptInfo 有提示作用
//也就是说当前的内表的路径如果已经满足了这个 PathKeys，那么就可以免掉内表的排序
innerkeys = make_inner_pathkeys_for_merge(root,
                                          cur_mergeclauses,
                                          outerkeys);

//根据外表的 PathKeys，产生 MergeJoin 路径的 PathKeys，提供给上层的操作借鉴使用
merge_pathkeys = build_join_pathkeys(root, joinrel, jointype,
                                     outerkeys);
```

我们给出的例子都比较简单，因为约束条件中的每个等值约束条件都是等价的，因此它的 innerkeys 和 outerkeys 是一样的，但如果在连接关系中有外连接，则可能 innerkeys 和 outerkeys 不一定相同。

8.5.1.3　MergeJoin 路径及代价

构建 MergeJoin 路径的过程本身比较简单，就是向 MergePath 结构体中填充已有的各种数据，由于 PathKeys 的不同会产生很多不同的 MergeJoin 路径，因此创建物理路径带来的消耗也是需要考虑的。PostgreSQL 数据库会在创建路径之前做预检查，对一些明显较差的路径在创建之前就将其淘汰。

Mergejoin 的代价计算分成了两个阶段。

- 阶段 1：计算初步的代价，这个代价就是用来筛选掉一些明显的代价较高的路径，这样能够节省查询优化的执行时间。

- 阶段 2：计算最终的代价，如果已经决定要创建 MergeJoin 路径，这时候就根据 MergeJoin 在查询执行器中的执行方法来计算它的各个步骤的代价。

在初步代价的计算过程中最重要的是排序带来的代价（cost_sort 函数），因为 MergeJoin 可能需要对内表和外表进行显式的排序（除非内表和外表路径已经有符合 Mergejoinable 的连接条件的 PathKeys），根据待排序元组的输入字节数（input_bytes）、输出字节数（output_bytes）及可使用的排序内存的大小（sort_mem_bytes），排序又分成了 3 种情况。

- output_bytes > sort_mem_bytes，也就是说排序的结果会有一部分保存到外存，这时候计算代价考虑使用外排序+多路归并的方法。

```
//外排序需要比较 N×log(N)次，这是 CPU 代价
startup_cost += comparison_cost * tuples * LOG2(tuples);

/* 应用换底公式，Compute logM(r) as log(r) / log(M) */
if (nruns > mergeorder)
   log_runs = ceil(log(nruns) / log(mergeorder));
else
   log_runs = 1.0;

//计算页面的访问次数
npageaccesses = 2.0 * npages * log_runs;
/* Assume 3/4ths of accesses are sequential, 1/4th are not */
startup_cost += npageaccesses *
   (seq_page_cost * 0.75 + random_page_cost * 0.25);
```

- input_bytes > sort_mem_bytes & output_bytes < sort_mem_bytes（LIMIT 子句也会产生这种情况），这时对应的情况可能是 TOP-K 的情况，也就是说只需要获得排序后的前 K 个元组，而这前 K 个元组是可以放到内存的，这时候计算代价考虑使用堆排序的方法，在内存中保持 TOP-K 个元组，算法的时间复杂度是 N×log(K)，PostgreSQL 考虑到代价曲线的连续性，计算代价时使用的是 N×log(2K)。
- 如果待排序的元组完全能加载到内存，则计算代价的时候考虑使用快速排序的方法，算法的时间复杂度是 N×log(N)。

初级代价中需要计算第一个匹配上连接条件的元组之前的元组所占的比例（outerstartsel 变量和 innerstartsel 变量）和最后一个匹配连接条件元组之前的元组所占的比例（outerendsel 变量和 innerendsel 变量），这里本来需要使用所有的 Mergejoinable 的连接条件来估算这些值，但是考虑到这个操作是一个比较低效的操作，因此 PostgreSQL 数据库只使用第一个连接条件来估算这些

值，我们用一个例子来说明一下 startsel 和 endsel 的含义，例如有两个表 A 和 B 的数据如下：

A 表：　1　2　3　4　5　6　7　8　9　10

B 表：　4　5　6　7　8　9　10　11　12　13

A 表作为外表需要到第 4 条记录的时候才能匹配到 B 表的第 1 条记录，也就是说 A 表的 outerstartsel 是 0.3（共 10 条数据，前面 3 条都没有匹配上），B 表作为内表第一条记录就能和 A 中的记录匹配上，因此 B 表的 innerstartsel 是 0。A 表的最后一条数据是 10，它能和 B 表的 10 匹配上，也就是说在 MergeJoin 的过程中，A 表能扫描到结尾，因此 A 表的 outerendsel 的值是 1，而 B 表只需要扫描到 10 就可以结束了，因为 MergeJoin 过程中有一方的数据结束了，另一方也就可以结束了，因此 B 表的 innerendsel 的值是 0.7（共 10 条记录，扫描到第 7 条）。需要注意的是，outerstartsel 和 innerstartsel 中至少有一个是 0，同样，outerendsel 和 innerendsel 中至少有一个是 1。通过 outerstartsel 或 innerstartsel 能够计算 MergeJoin 的启动代价，通过 outerendsel 或者 innerendsel 可以计算 MergeJoin 的执行代价。

初始代价是在路径建立之前进行估计的，这样在获得初始代价之后，就可以决定是否建立这个路径，如果需要建立这个路径，就开始创建 MergeJoin 路径，需要注意的是并行 MergeJoin 路径的并行度取决于外表的并行度。

在创建 MergeJoin 路径之后还需要精确地计算 MergeJoin 路径的最终代价，这个过程复用了初始代价的计算结果，然后继续计算路径中的表达式代价、物化代价等。

```
void
final_cost_mergejoin(PlannerInfo *root, MergePath *path, JoinCostWorkspace *workspace,
                JoinPathExtraData *extra)
{
   ……

   //对内表的行数做校正
   if (inner_path_rows <= 0 || isnan(inner_path_rows))
      inner_path_rows = 1;

   //获得当前表的行数
   if (path->jpath.path.param_info)
      path->jpath.path.rows = path->jpath.path.param_info->ppi_rows;
   else
      path->jpath.path.rows = path->jpath.path.parent->rows;

   //如果是并行路径，校正为每个子路径的行数
```

```
    if (path->jpath.path.parallel_workers > 0)
        ......

    //计算(Mergejoinable)连接条件的表达式代价
    cost_qual_eval(&merge_qual_cost, mergeclauses, root);
    //计算所有的约束条件的表达式代价
    cost_qual_eval(&qp_qual_cost, path->jpath.joinrestrictinfo, root);
    //获得除(Mergejoinable)的连接条件之后，剩余的约束条件的表达式代价
    qp_qual_cost.startup -= merge_qual_cost.startup;
    qp_qual_cost.per_tuple -= merge_qual_cost.per_tuple;
    //注意：以上计算的代价都是针对单条元组的，它只是为将来的整体的表达式代价计算做准备
    ......
}
```

在 MergeJoin 路径执行时，会对已经排序的内外表的元组进行归并操作，假如内表中有重复值，则需要对同样的重复值进行多次扫描，如图 8-5 所示，在外表的第一个 5 匹配了内表的所有的 5 之后，外表的第二个 5 还需要重新扫描一次内表的所有的 5，因此 PostgreSQL 在内表的第一个 5 的地方做了一个标记（Mark），外表向前移动一条元组，如果还是 5，内表就做一个归位（restore）操作，在这种情况下，内表的一条元组可能要被扫描很多次。

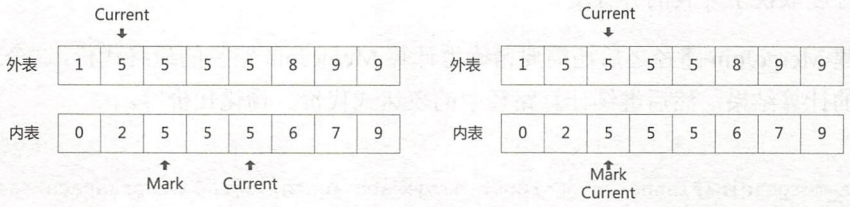


图 8-5　MergeJoin 中的内表重复扫描

假设外表去重之后有 k 个值，就可以按照这 k 个值将外表分成 k 个区间，每个区间内的值都是相等的，每个区间中的值的个数是 M_k，同样假设内表也分成 j 个区间，每个区间内的值的个数是 N_j，**并且假设内表和外表所有的区间都能匹配上**，那么重复扫描的元组数是：

$$\begin{aligned} rescannedtuples &= (M_1 - 1)\times N_1 + (M_2 - 1)\times N_2 + \cdots \\ &= (M_1\times N_1 + M_2\times N_2 + \cdots) - (N_1 + N_2 + \cdots) \\ &= \text{size of join} - \text{size of inner relation} \end{aligned}$$

重复扫描的情况并不是在所有情况下都存在的，例如在内表路径具有唯一性的情况下，就无须重复扫描内表，因此可以忽略这部分代价，另外如果连接类型是 SemiJoin 或者 AntiJoin，也是无须重复扫描的。请注意在判断内表是否具有唯一性、连接类型是否是 SemiJoin 或者

AntiJoin 的同时，源代码中还判断了(list_length(path->jpath.joinrestrictinfo) == list_length(path->path_mergeclauses)))，这个判断的意思是要求所有的连接条件都必须是 MergeJoinable 的连接条件，这是因为假如除了 Mergejoinable 的连接条件之外还有其他的连接条件，那么满足 Mergejoinable 的连接结果不一定能满足其他的连接条件，因此可能还需要对内表重复扫描。另外如果外表的路径具有唯一性，实际上也无须重复扫描。

下面我们来分析一下 MergeJoin 是如何计算代价的。

```
void
final_cost_mergejoin(PlannerInfo *root, MergePath *path,
                JoinCostWorkspace *workspace,
                JoinPathExtraData *extra)
{
   ……

   //SemiJoin、AntiJoin、Inner UniquePath 都属于匹配到一条即可返回
   if ((path->jpath.jointype == JOIN_SEMI ||
        path->jpath.jointype == JOIN_ANTI ||
        extra->inner_unique) &&
        //所有的连接条件必须是 MergeJoinable 的连接条件
       (list_length(path->jpath.joinrestrictinfo) ==
        list_length(path->path_mergeclauses)))
      path->skip_mark_restore = true;
   else
      path->skip_mark_restore = false;

   //能匹配上（Mergejoinable）连接条件的结果集数量
   mergejointuples = approx_tuple_count(root, &path->jpath, mergeclauses);

   //如果外表是 UniquePath，那么它相邻的两个元组肯定是不等值的，因此无须重复扫描
   //如果内表满足具有唯一性或连接类型是 SemiJoin 或者 AntiJoin 情况，那么也无须重复扫描
   if (IsA(outer_path, UniquePath) ||path->skip_mark_restore)
      rescannedtuples = 0;
   else
   {
      //计算需要重复扫描的元组数量
      rescannedtuples = mergejointuples - inner_path_rows;
      if (rescannedtuples < 0)
         rescannedtuples = 0;
   }
   //1.0 是内表必须扫描的次数
```

```
    //(rescannedtuples / inner_path_rows)是内表重复扫描的元组所占的比例
    rescanratio = 1.0 + (rescannedtuples / inner_path_rows);

    //考虑到重复扫描，内表的扫描代价需要调整（这时不物化内表）
    bare_inner_cost = inner_run_cost * rescanratio;

    //内表物化之后，假设获取元组的代价只有 cpu_operator_cost
    mat_inner_cost = inner_run_cost +
        cpu_operator_cost * inner_path_rows * rescanratio;

    //没有重复扫描，也就无须对内表物化
    if (path->skip_mark_restore)
        path->materialize_inner = false;
    //如果物化之后的代价小于非物化的代价，那么考虑将内表物化
    else if (enable_material && mat_inner_cost < bare_inner_cost)
        path->materialize_inner = true;

    //innersortkeys 代表的是内表不是显式排序的，
    //例如内表路径是 B 树索引扫描路径，且它符合（Mergejoinable）连接条件的键值
    //ExecSupportsMarkRestore 负责检查那些不是显式排序而且不支持重复扫描的路径
    //例如内表本身就是 MergeJoin 路径，它产生的结果是有序的，但它不支持重复扫描
    else if (innersortkeys == NIL && !ExecSupportsMarkRestore(inner_path))
        path->materialize_inner = true;

    //如果内表是显式排序的，而且排序结果需要交换到外存，那么也有必要物化
    else if (enable_material && innersortkeys != NIL && relation_byte_size(inner_path_rows,
          inner_path->pathtarget->width) >  (work_mem * 1024L))
        path->materialize_inner = true;
    else
        path->materialize_inner = false;

    //如果要将内表物化，计入物化的代价，否则计入不物化的代价
    if (path->materialize_inner)
        run_cost += mat_inner_cost;
    else
        run_cost += bare_inner_cost;

    //计算（Mergejoinable)的连接条件产生的启动代价
    startup_cost += merge_qual_cost.startup;
    startup_cost += merge_qual_cost.per_tuple *  (outer_skip_rows + inner_skip_rows *
rescanratio);
```

```
    //计算 (Mergejoinable)的连接条件产生的执行代价
    run_cost += merge_qual_cost.per_tuple * ((outer_rows - outer_skip_rows) +
        (inner_rows - inner_skip_rows) * rescanratio);

    //计算除 (Mergejoinable) 连接条件之外的其他约束条件带来的启动代价和执行代价
    startup_cost += qp_qual_cost.startup;
    cpu_per_tuple = cpu_tuple_cost + qp_qual_cost.per_tuple;
    run_cost += cpu_per_tuple * mergejointuples;

    //计算投影列中的表达式的启动代价和执行代价
    startup_cost += path->jpath.path.pathtarget->cost.startup;
    run_cost += path->jpath.path.pathtarget->cost.per_tuple * path->jpath.path.rows;

    path->jpath.path.startup_cost = startup_cost;
    path->jpath.path.total_cost = startup_cost + run_cost;
}
```

8.5.2　match_unsorted_outer 函数

sort_inner_and_outer 函数负责生成显式排序的 MergeJoin 路径，因此它只选择了待连接的两个 RelOptInfo 中的代价最低的路径（cheapest_total_path，并行路径建立时则考虑了外表的最低代价的并行路径）参与 MergeJoin 路径的生成，由于需要显式排序，因此最低启动代价的路径没有什么用。

而 match_unsorted_outer 则负责考虑那些外表已经有序的，对内表需要显式排序的情况，例如外表是 B 树索引扫描路径，它的扫描结果就是有序的。

在早期的 PostgreSQL 数据库中还存在 match_unsorted_inner 函数，后来这个函数被废弃了，原因是 add_paths_to_joinrel 函数已经负责考虑过交换两个待连接的 RelOptInfo 的顺序，这样同一个 RelOptInfo 在 match_unsorted_outer 函数中会各有一次机会做外表和内表，如果还有 match_unsorted_inner 函数生成的路径，就会和 match_unsorted_outer 函数生成的路径重复。

8.5.2.1　路径生成流程

如果外表是有序的，那么无论采用 NestloopJoin 连接方法还是 MergeJoin 连接方法，连接的结果也会和外表的顺序保持一致，所以在 match_unsorted_outer 函数中，主要考虑生成 NestloopJoin 连接路径和 MergeJoin 连接路径。

对于生成 MergeJoin 路径，在 sort_inner_and_outer 函数中假定内表和外表都是无序的，必须显式地对两个表排序，因此它是通过 MergeJoinable 的连接条件来生成 PathKeys，然后不断地

调整 PathKeys 中的 PathKey 的顺序来获得不同顺序的 PathKeys，然后再根据不同顺序的 PathKeys 来决定内表的 innerkeys 和外表的 outerkeys。而在 match_unsorted_outer 函数中，假定外表路径是有序的，它是按照外表的 PathKeys 反过来排序连接条件的（外表的 PathKeys 直接就可以作为 outerkeys 使用），查看连接条件中有哪些是和当前的 PathKeys 匹配的，把匹配的连接条件筛选出来（注意：因为用的是外表的 PathKeys，因此并不是所有的 MergeJoinable 约束条件都可以用上），然后再参照匹配出来的连接条件生成内表需要显式排序的 innerkeys，对比情况如图 8-6 所示。

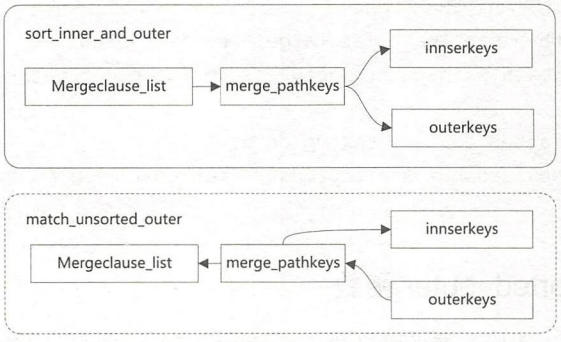


图 8-6　PathKeys 生成的不同方式

在有了 PathKeys 和 Mergejoinable 连接条件之后，MergeJoin 路径的建立和 sort_inner_and_outer 函数中类似，这里不再赘述。

在生成 NestloopJoin 连接路径的过程中，针对外表的每一条路径，内表都需要考虑：

- 最低代价的路径。
- 参数化路径。
- 最低代价路径产生的物化路径。

另外还考虑产生于 NestloopJoin 和 MergeJoin 的并行连接路径，需要注意的是并行 NestloopJoin 路径的内表看似只考虑了参数化路径，但最低代价的非参数化路径也在 innerrel->cheapest_parameterized_paths 中保留了一份，所以实际上也考虑了最低代价的非参数化路径。

8.5.2.2　NestloopJoin 代价计算

我们已经介绍过 MergeJoin 的代价计算的过程，下面介绍一下 NestloopJoin 的代价计算模型。

NestloopJoin 的代价计算和 MergeJoin 的代价计算类似，都分成了初级代价（initial_cost_nestloop 函数）和最终代价（final_cost_nestloop 函数），初级代价用于决定是否创建路径，最

终路径则用于筛选已经创建的路径。

NestloopJoin 连接方法的时间复杂度是 $O(m \times n)$，外表的每一条元组都会扫描一次内表（如果内表路径是类似于索引扫描这种路径，NestloopJoin 的时间复杂度会低一些），因此内表会被多次扫描，在代价计算的过程中需要考虑到对一个表如果连续地多次扫描，那么第一次扫描的代价和之后多次扫描的代价是不同的。例如对于一个排序路径，在第一次执行的时候会有排序产生的启动代价，在第一次扫描之后排序已经完成并且保留了排序结果，在第二次扫描的时候，就可以直接利用第一次扫描的排序结果，而无须再次排序，因此第二次扫描就不用计入排序带来的启动代价，cost_rescan 函数主要分析了类似的这种情况，用于计算第二次及之后的内表扫描的代价。

需要注意的是，第一次扫描和之后的扫描所产生的磁盘 IO 实际上也会有很大的不同，我们做一个极端的假设，在第一次扫描的时候内表的所有页面都保存在磁盘上，扫描的过程中需要将所有的页面都加载到内存中，但是在第二次扫描时，由于第一次扫描已经加载了页面，因此就不会产生磁盘 IO（假设页面加载进内存后都没有被换出），不过 cost_rescan 函数没有考虑这种情况，这也是 PostgreSQL 数据库的查询优化器可以改进的地方。

在初级代价的计算中还考虑了 SemiJoin、AntiJoin、UniquePath 的情况，针对这 3 种情况，在初级代价中没有计算它们的执行代价，它们的执行代价是在最终代价的阶段计算的。

初级代价的计算模型如下（不考虑 SemiJoin、AntiJoin、UniquePath 的情况）：

$$\text{初级启动代价} = \text{外表路径的启动代价} + \text{内表路径的启动代价}$$

$$\begin{aligned}\text{初级执行代价} = {} & \text{外表的执行代价} + \text{内表第一次扫描的执行代价} + \\ & \text{内表多次扫描的启动代价} \times (\text{外表估计行数} - 1) + \\ & \text{内表多次扫描的执行代价} \times (\text{外表估计行数} - 1)\end{aligned}$$

在最终代价的计算过程中，有个很重要的变量是连接关系产生的结果集的数量（在没有约束条件的情况下），对于普通的连接关系，它们产生的结果集的数量就是 $m \times n$，也就是内表元组的数量×外表元组的数量，获得的这个结果集数量的主要原因是要计算路径中的表达式代价，因此这里的结果集数量不是约束条件过滤之后的结果集数量，而是外表和内表进行笛卡儿积产生的结果数量，我们在这个基础之上进行约束条件的表达式代价的计算、投影表达式代价的计算等。另外，如果内表是索引扫描路径，它有两种可能的情况，一种情况是把索引当作普通表对待，这样它的结果集仍然是做卡氏积得到 $m \times n$，另一种情况是内表路径是参数化路径，参数化的连接条件下降到索引扫描上了，在 final_cost_nestloopjoin 函数中对这种情况的内表行数已经进行了调整。

严禁网络传播本PDF，违者责任自负！

```
if (path->path.param_info)
   path->path.rows = path->path.param_info->ppi_rows;
else
   path->path.rows = path->path.parent->rows;
```

在 SemiJoin、AntiJoin、UniquePath 三种情况下，由于对于外表的一条元组，在内表匹配到一个元组即可，因此这里需要利用在 compute_semi_anti_join_factors 函数中已经计算好的两个变量来估计结果集的数量。其中 outer_match_frac 的含义是指外表中能与内表匹配上的元组数量占外表总元组数量的比例，match_count 代表的是对于一条外表元组平均要扫描多少元组才能在内表中找到第一条匹配的元组，这样可以得到 SemiJoin、AntiJoin、UniquePath 三种情况下和匹配项相关的连接的结果集的数量。

如果在 SemiJoin、AntiJoin、UniquePath 三种情况下，外表中的元组和内表没有匹配的元组需要怎么计算扫描的次数呢？如果内表是索引扫描，有没有匹配项一查便知，因此这部分就不再计算了；如果内表是其他扫描方式，则需要扫描整个内表才能确定是否有匹配项，这部分必须计入结果集的数量中。

```
if (path->jointype == JOIN_SEMI || path->jointype == JOIN_ANTI ||extra->inner_unique)
{
   //外表可以和内表的某些元组匹配，那么需要探测元组形成的结果集数量
   //outer_matched_rows * inner_path_rows 代表的是外表匹配元组和内表的卡氏积数量
   //inner_scan_frac 相当于一条外表匹配元组会对应多少条内表元组的倒数
   //这样就计算出了一条外表元组匹配到第一条元组时需要匹配的元组数
   ntuples = outer_matched_rows * inner_path_rows * inner_scan_frac;
   if (has_indexed_join_quals(path))
      { //如果是索引扫描，内表没有匹配上元组一查便知，不会进行表达式代价计算}
   else
      {//如果不是索引扫描，需要遍历整个内表才能知道是不是能匹配上
      ntuples += outer_unmatched_rows * inner_path_rows;  }
}
else
{
   //卡氏积结果集数量
   ntuples = outer_path_rows * inner_path_rows;
}
```

在最终代价中对 SemiJoin、AntiJoin、UniquePath 三种情况的执行代价（run_cost）进行了计算（初级代价的时候没有计算），由于内表路径的类型不同，产生的结果集数量也不同，那么计算代价的方式也不同。

严禁网络传播本PDF，违者责任自负！

```
if (has_indexed_join_quals(path))
{
    //因为找到一条满足约束条件的内表元组即可，使用 inner_scan_frac 校正代价
    //第一次扫描内表的时候使用 innser_run_cost
    run_cost += inner_run_cost * inner_scan_frac;
    //第一次扫描之后的扫描代价用 inner_rescan_run_cost 计算
    if (outer_matched_rows > 1)
        run_cost += (outer_matched_rows - 1) * inner_rescan_run_cost * inner_scan_frac;

    //如果是索引扫描，且内表没有匹配项，这个代价是比较低的
    //inner_rescan_run_cost / inner_path_rows 是 rescan 过程中一条元组的代价
    run_cost += outer_unmatched_rows *
        inner_rescan_run_cost / inner_path_rows;
}
else
{
    /* First, count all unmatched join tuples as being processed */
    ntuples += outer_unmatched_rows * inner_path_rows;

    /* Now add the forced full scan, and decrement appropriate count */
    //这种情况下，在 Nestloop Join 结束之后都不一定把内表完全扫描一遍
    //一个极端的例子是外表的元组全部是 1，内表的元组第一条是 1，那么每次内表只扫描第一条就能匹配上
    //但这里做一个保守的假设，就是内表至少要全表扫描 1 次
    run_cost += inner_run_cost;
    //强制增加的这一次全表扫描，如果外表有和内表匹配不上的元组，就记到它的头上
    if (outer_unmatched_rows >= 1)
        outer_unmatched_rows -= 1;
    else
        //如果外表没有和内表匹配不上的元组，那么只能记到外表匹配元组的账上
        //（这种情况只有在外表匹配的内表元组恰好是内表的最后一条时才可能发生）
        outer_matched_rows -= 1;

    //和内表相匹配元组连接产生的代价
    if (outer_matched_rows > 0)
        run_cost += outer_matched_rows * inner_rescan_run_cost * inner_scan_frac;

    //和内表不匹配的元组连接产生的代价（第一次扫描的代价已经在上面计入了这里使用 rescan 扫描的代价）
    if (outer_unmatched_rows > 0)
        run_cost += outer_unmatched_rows * inner_rescan_run_cost;
}
```


严禁网络传播本PDF，违者责任自负！

8.5.3 hash_inner_and_outer 函数

顾名思义，hash_inner_and_outer 函数的主要作用是建立 HashJoin 的路径，在 distribute_restrictinfo_to_rels 函数中已经判断过一个约束条件是否适用于 HashJoin，如果适用于 HashJoin 还会记录对应的操作符的 Oid，另外还需要区分一个约束条件是连接条件还是过滤条件，在外连接的情况下，如果约束条件的 is_push_down==true，就代表它是一个过滤条件，过滤条件不能用作连接条件。HashJoin 要建立哈希表，至少有一个适用于 HashJoin 的连接条件存在才能使用 HashJoin，否则无法创建哈希表。

HashJoin 创建的过程中主要考虑了外表的最低启动代价路径（cheapest_startup_path）和最低总代价的路径（cheapest_total_path），内表主要考虑了最低总代价路径（cheapest_total_path）。那么内表为什么没有考虑最低启动代价的路径呢？这是因为如果对数据进行了重新分布，那么最低启动代价的路径的意义就不大了，例如如果内表要进行唯一化，那么无论是最低启动代价路径还是最低总代价路径，在唯一化之后原来的启动代价都没有什么意义了，因此这时候就可以只考虑最低总代价路径。HashJoin 的内表由于要把数据重新分布成哈希表，因此内表选择最低启动代价也没有什么意义，所以内表只选择最低启动代价路径就可以了。我们先来分析一下非并行 HashJoin 路径创建过程，它的内表和外表的路径选择的流程如下：

```
if (jointype == JOIN_UNIQUE_OUTER)
{
    //外表无须选择最低启动代价的路径
    外表：cheapest_total_outer,
    内表：cheapest_total_inner,
}
else if (jointype == JOIN_UNIQUE_INNER)
{
    //内表无须选择最低启动代价的路径
    外表：cheapest_total_outer 和 cheapest_startup_outer
    内表：cheapest_total_outer
}
else
{
    //情况 1，外表选择最低启动代价的路径
    外表：cheapest_startup_outer
    内表：  cheapest_total_inner

    //情况 2：需要注意的是 cheapest_parameterized_paths 中包含最低总代价的路径
    foreach(lc1, outerrel->cheapest_parameterized_paths)
    {
```


严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
        foreach(lc2, innerrel->cheapest_parameterized_paths)
        {
            //最低总代价路径和参数化路径组合形成 HashJoin 路径
        }
    }
}
```

目前 HashJoin 的并行路径分成了两种，它们的区别主要在于哈希表是每个工作进程都自己生成还是所有的工作进程共享一份哈希表，我们通过两个执行计划来看一下它们的区别。

```
--不共享哈希表，每个工作进程都有一个哈希表
postgres=# set enable_parallel_hash=false;
SET
postgres=# EXPLAIN SELECT * FROM TEST_A,TEST_D WHERE TEST_A.a = TEST_D.a;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Gather  (cost=154398.98..482289.82 rows=3433947 width=32)
   Workers Planned: 2
   -> Hash Join  (cost=153398.98..481255.48 rows=1430811 width=32)
        Hash Cond: (test_a.a = test_d.a)
        -> Parallel Seq Scan on test_a  (cost=0.00..269548.92 rows=11733192 width=16)
        -> Hash  (cost=153252.88..153252.88 rows=11688 width=16)
             -> Seq Scan on test_d  (cost=0.00..153252.88 rows=11688 width=16)
(7 rows)

--并行创建哈希表，并共享哈希表
postgres=# set enable_parallel_hash=true;
SET
postgres=# EXPLAIN SELECT * FROM TEST_A,TEST_D WHERE TEST_A.a = TEST_D.a;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Gather  (cost=154245.58..473790.01 rows=3433947 width=32)
   Workers Planned: 2
   -> Parallel Hash Join  (cost=153245.58..472755.67 rows=1430811 width=32)
        Hash Cond: (test_a.a = test_d.a)
        -> Parallel Seq Scan on test_a  (cost=0.00..269548.92 rows=11733192 width=16)
        -> Parallel Hash  (cost=153184.70..153184.70 rows=4870 width=16)
             -> Parallel Seq Scan on test_d  (cost=0.00..153184.70 rows=4870 width=16)
(7 rows)
```

需要注意的是在执行计划中，HashJoin 的标识是不同的，如果每个工作进程都自建一个哈希表，那么 PostgreSQL 不认为这个 HashJoin 是一个真正的并行路径，它的路径的 parallel_aware

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

值是 false，执行计划中 HashJoin 前面也没有 Parallel 标识，而另一种情况 HashJoin 路径的 parallel_aware 变量值是 true，这时候它的路径名是 Parallel Hash Join。PostgreSQL 分别称这两种情况为 parallel-oblivious hash join 和 parallel-aware hash join，它们主要是通过 try_partial_hashjoin_path 函数中的参数 parallel_hash 来区分，parallel_hash 参数的值为 true 的时候代表创建的是共享哈希表（parallel-aware hash join）的并行 HashJoin 路径。

HashJoin 路径的代价和 NestloopJoin、MergeJoin 类似同样采用两阶段的方式进行计算，第一个节点计算初级代价用来决定是否创建路径，第二个阶段计算最终代价用来对创建的路径进行筛选。

初级代价是通过 initial_cost_hashjoin 函数来做的，共计算了 3 部分代价。

第一部分，内表的总代价（因为创建哈希表需要对数据进行重分布）和外表的启动代价都计入 HashJoin 的启动代价，外表的执行代价（外表的总代价-外表的启动代价）计入执行代价。

第二部分，通过内表的元组创建哈希表的代价是启动代价，内表的每个元组所占的代价包括处理一个元组的基本代价和应用在元组上的哈希条件的代价（cpu_operator_cost * num_hashclauses + cpu_tuple_cost），再乘以内表的行数（inner_path_rows）就是创建哈希表的总代价。外表的每个元组从哈希表检测是否匹配哈希条件的代价是执行代价，其中（cpu_operator_cost * num_hashclauses）是每个元组匹配的代价，outer_path_rows 是外表的行数。

第三部分，如果内表的数据比较多，哈希表无法全部加载在内存，需要将内表数据分散到多个 batch，外表也会有对应数量的 batch，内表的第一个 batch 保存在内存中，外表的数据如果匹配到了第一个 batch 中（已经加载到内存的 batch）的数据，那么可以直接产生匹配之后的连接结果，如果匹配到了不是第一个 batch（即匹配到内表没有加载到内存的 batch），那么外表数据保存在外表的 batch 中（内表的 batch 和外表的 batch 一一对应），这样就能够保证匹配上的数据都保存在一一对应的内表 batch 和外表 batch 中，因此在处理完第一个内表的 batch 之后，可以切换到下一个内表 batch 和外表 batch 直接生成连接结果，这个过程需要将数据从外存写和读各一次，内表建立 batch 的过程（同时建立哈希表）是将内表数据写入 batch 的过程，这部分代价计入启动代价，其他的部分计入执行代价。

```
if (numbatches > 1)
{
  double     outerpages = page_size(outer_path_rows,
                                    outer_path->pathtarget->width);
  double     innerpages = page_size(inner_path_rows,
                                    inner_path->pathtarget->width);
```


非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
    //建立哈希表的过程中也会建立 batch，内表数据写入 batch 的代价计入启动代价
    startup_cost += seq_page_cost * innerpages;
    //内表 batch 的数据会读取出来再次匹配
    //外表的数据会建立 batch 写一次，和内表的 batch 匹配时读一次，共计两次
    run_cost += seq_page_cost * (innerpages + 2 * outerpages);
}
```

HashJoin 的最终代价的计算主要还是计算哈希条件及减去哈希条件之后剩余的约束条件的表达式代价。

```
//计算哈希表中每个桶的平均元组数，如果内表具有唯一性，假设它和桶数相等
if (IsA(inner_path, UniquePath))
    innerbucketsize = 1.0 / virtualbuckets;
else
{
    //根据哈希条件（连接条件）计算 innerbucketsize（基于统计信息）
    //每个哈希条件计算一个 innerbucketsize，选择最小的那个作为计算代价的值
}

//计算哈希条件的表达式代价
cost_qual_eval(&hash_qual_cost, hashclauses, root);
//计算所有约束条件的表达式代价
cost_qual_eval(&qp_qual_cost, path->jpath.joinrestrictinfo, root);
//下面会单独统计哈希条件的表达式代价，因此在约束条件表达式代价里先减去这部分代价
qp_qual_cost.startup -= hash_qual_cost.startup;
qp_qual_cost.per_tuple -= hash_qual_cost.per_tuple;

/* CPU costs */
if (path->jpath.jointype == JOIN_SEMI ||
    path->jpath.jointype == JOIN_ANTI ||
    extra->inner_unique)
{
    double      outer_matched_rows;
    Selectivity inner_scan_frac;

    //首先，计算外表和内表能够匹配的元组的那部分的代价，
    //可参考 NestloopJoin 的代价计算的部分
    outer_matched_rows = rint(outer_path_rows * extra->semifactors.outer_match_frac);
    inner_scan_frac = 2.0 / (extra->semifactors.match_count + 1.0);
```


非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
    startup_cost += hash_qual_cost.startup;
    //执行代价的计算中考虑了“只需匹配一条元组即可（inner_scan_frac）”
    //0.5 是默认的估计值
    run_cost += hash_qual_cost.per_tuple * outer_matched_rows *
        clamp_row_est(inner_path_rows * innerbucketsize * inner_scan_frac) * 0.5;

    //然后，计算不能匹配的部分的代价
    //这里的 0.05 是 0.1（PostgreSQL 代码注释中的十分之一）和 0.5（默认估计值）的乘积
    run_cost += hash_qual_cost.per_tuple *
      (outer_path_rows - outer_matched_rows) *
      clamp_row_est(inner_path_rows / virtualbuckets) * 0.05;

    if (path->jpath.jointype == JOIN_SEMI)
        hashjointuples = outer_matched_rows;
    else
        hashjointuples = outer_path_rows - outer_matched_rows;
}
else
{
    //计算哈希条件的表达式代价
    startup_cost += hash_qual_cost.startup;
    //其中，inner_path_rows * innerbucketsize 表示一个 hash bucket 中的元组数
    //实际上一条外表的元组不一定把所有的哈希条件都执行一次
    //但这里没有好的模型来调整这个不确定的代价，默认使用 0.5 做调整
    run_cost += hash_qual_cost.per_tuple * outer_path_rows *
    clamp_row_est(inner_path_rows * innerbucketsize) * 0.5;

    //计算匹配哈希条件的元组有多少个
    hashjointuples = approx_tuple_count(root, &path->jpath, hashclauses);
}

//上面计算的是哈希条件的代价，hashjointuples 是通过哈希表检测的元组数
//通过哈希表检测的元组还需要通过约束条件（已减去哈希条件代价）检测，计算这部分代价
startup_cost += qp_qual_cost.startup;
cpu_per_tuple = cpu_tuple_cost + qp_qual_cost.per_tuple;
run_cost += cpu_per_tuple * hashjointuples;

//计算投影产生的表达式代价
startup_cost += path->jpath.path.pathtarget->cost.startup;
run_cost += path->jpath.path.pathtarget->cost.per_tuple * path->jpath.path.rows;
```


非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
path->jpath.path.startup_cost = startup_cost;
path->jpath.path.total_cost = startup_cost + run_cost;
```

8.6　路径的筛选

从连接路径代价计算的过程可以看出，它的代价计算目前都分成了两个部分：初级代价计算和最终代价计算，初级代价是用来决定是否创建这个路径，是一个预筛选的过程，目前只针对连接路径有这样的预筛选，因为对于基表来说，扫描路径的种类是有限的，而且这时还是需要尽可能多地考虑扫描路径，这样才能给上层的连接路径打好基础，因此在扫描路径生成阶段没有引入预筛选（在位图扫描等扫描路径阶段引入了并行路径的预筛选，但是道理基本相同）。而在连接路径阶段的可能性通常是较多的，而且可能会生成一些明显比较差的路径，这时候预筛选能帮助我们做一个基本的检查，能够节省生成执行计划的时间，因为如果生成执行计划的时间太长，即使选出了“很好”的执行计划，也是不能接受的。

预筛选通常使用的是初级代价，这是一个不完整的代价，因为从代价计算的过程可以看出，最终代价的计算都是在初级代价的基础上增加新的代价，例如增加表达式代价等，因此如果一个不完整的代价和已有路径的整体代价相比都没有任何优势，那么这个连接路径被预筛选掉也就不“冤枉”了。

通过了预筛选之后，物理路径就会被创建并且计算路径的最终代价，这个代价才是代价模型要计算的整体代价，在 add_path 函数中，还需要根据这个整体代价再做一次检查，和预筛选不同的是，预筛选只决定新路径创建不创建，而 add_path 函数在新路径特别优秀的情况下，还会淘汰一些老路径。

PostgreSQL 数据库单独实现了关于并行路径筛选的接口，使用 add_partial_path_precheck 函数做预筛选，使用 add_partial_path 函数做路径的最终筛选。

add_path 函数主要将新路径和 RelOptInfo->pathlist 来进行比较，而 add_partial_path 函数主要将新路径和 RelOptInfo->partial_pathlist 进行比较。

那么路径筛选的依据是什么呢？它主要取决于以下几个因素：

- 路径的启动代价，这是一个可选项，因为通常在 SQL 语句中含有 LIMIT 子句时，启动代价才有用，因此在路径筛选过程中是否考虑启动代价的非常重要的因素就是 tuple_fraction 是否有值。
- 路径的总代价，路径的总代价是最重要的一个指标。

非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

- 两个路径的 PathKeys 是不是具有可比性，如果有，哪个 PathKeys 更好。
- 两个路径返回的行数哪个少，返回行数少说明它具有更好的过滤性，给上层的操作减少计算量。
- 两个路径是否并行安全（parallel_safe），安全的好于非安全的（在源代码中判断 parallel_safe 时需要注意比较的符号是>还是>=，这里有细微的差别）。
- 如果是参数化路径，它们是否需要同样的“参数生产者”。

和代价有关的筛选主要采用的是 compare_path_costs_fuzzily 函数（宏）来实现的，它设定了一个值 STD_FUZZ_FACTOR（=1.01）来决定两个路径的代价哪个更有优势，例如 A > (B * STD_FUZZ_FACTOR)，则说明 B 的代价是优于 A 的。

基于代价的路径筛选对照如表 8-2 所示。

表 8-2　基于代价的路径筛选对照表

总代价比较	启动代价比较（假设需要比较）	比较结果
总代价A优于B	启动代价B优于A	COSTS_DIFFERENT：互相不占优势
	启动代价A优于B	COSTS_BETTER1：A优于B
总代价B优于A	启动代价B优于A	COSTS_BETTER2：B优于A
	启动代价A优于B	COSTS_DIFFERENT：互相不占优势
总代价A和B差不多	启动代价A优于B	COSTS_BETTER1：A优于B
	启动代价B优于A	COSTS_BETTER2：B优于A
	其他	COSTS_EQUAL：A ≈ B

PathKeys 是通过 compare_pathkeys 函数进行比较的，它的方法是逐个地比较每个 PathKey。

- 如果 A 和 B 中出现不同的 PathKeys，那么它们互相不占优势 =PATHKEYS_DIFFERENT。
- 如果 A 比 B 长，且公共部分相等，那么 A 优于 B = PATHKEYS_BETTER1。
- 如果 B 比 A 长，且公共部分相等，那么 B 优于 A = PATHKEYS_BETTER2。
- 如果 A 和 B 完全相同，那么 A 和 B 相等 =PATHKEYS_EQUAL。

路径预筛选（add_path_precheck 函数）主要是依据初级代价和 RelOptInfo->pathlist 中的所有路径逐个比较，如果要创建的路径比任何一个已有的路径相比有总代价差，且启动代价差、PathKeys 差、“参数的生产者”相同，那么这个路径的创建就可以避免了，也就是说这个路径和 RelOptInfo->pathlist 中的某一个路径相比没有任何优势，反过来说，只要它和任何路径相比都有一点优势，那么它就通过预筛选，可以创建这个路径。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

并行路径的预筛选函数 add_partial_path_precheck 在 add_path_precheck 函数的基础上又做了增强，它先检查了 RelOptInfo-> partial_pathlist，期望能够通过快速地比较 PathKeys 和总代价能得到一个确定的答案，如果要创建的并行路径明显比较差或者明显非常好，都可以不再进行 add_path_precheck，直接就可以决定不创建或者创建这个路径。

连接路径被创建之后会计算最终代价，这也是连接路径的真正总代价，总代价计算出来之后，就可以进入 add_path 函数或者 add_partial_path 函数开始决定自己是否可以加入 RelOptInfo->pathlist 或者 RelOptInfo->partial_pathlist 中，同时还可以决定是否需要淘汰明显不好的路径。

```
//判断条件共 5 个因素：代价、PathKeys、路径参数、结果集行数、并行安全性
//COSTS_DIFFERENT：接受新路径，保留旧路径
if (costcmp != COSTS_DIFFERENT)
{
   /* Similarly check to see if either dominates on pathkeys */
   List      *old_path_pathkeys;

   old_path_pathkeys = old_path->param_info ? NIL : old_path->pathkeys;
   keyscmp = compare_pathkeys(new_path_pathkeys,
                        old_path_pathkeys);
   if (keyscmp != PATHKEYS_DIFFERENT)
   {
      switch (costcmp)
      {
         case COSTS_EQUAL:
            outercmp = bms_subset_compare(PATH_REQ_OUTER(new_path),
                                          PATH_REQ_OUTER(old_path));
            if (keyscmp == PATHKEYS_BETTER1)
            {
               if ((outercmp == BMS_EQUAL ||
                    outercmp == BMS_SUBSET1) &&
                   new_path->rows <= old_path->rows &&
                   new_path->parallel_safe >= old_path->parallel_safe)
                   //新路径：代价和老路径相似，PathKeys 要“长”，需要的路径参数少
                   //结果集行数少，比旧路径并行安全，接受新路径，放弃旧路径
                   remove_old = true;    /* new dominates old */
            }
            else if (keyscmp == PATHKEYS_BETTER2)
            {
               if ((outercmp == BMS_EQUAL ||
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
            outercmp == BMS_SUBSET2) &&
           new_path->rows >= old_path->rows &&
           new_path->parallel_safe <= old_path->parallel_safe)
           //新路径：代价和老路径相似，PathKeys 要“短”，需要的路径参数多
           //结果集行数多，不如旧路径并行安全，不接受新路径，保留旧路径
           accept_new = false; /* old dominates new */
    }
    else    /* keyscmp == PATHKEYS_EQUAL */
    {
       if (outercmp == BMS_EQUAL)
       {
          //到这里，新旧路径的代价、PathKeys、路径参数均相同或相似
          //如果新路径并行安全，选择接受新路径，放弃旧路径
          if (new_path->parallel_safe >
             old_path->parallel_safe)
             remove_old = true;    /* new dominates old */
          //如果新路径并行不如旧路径安全，选择不接受新路径，保留旧路径
          else if (new_path->parallel_safe <
                old_path->parallel_safe)
             accept_new = false; /* old dominates new */
          //如果新路径返回的行数少，选择接受新路径，放弃旧路径
          else if (new_path->rows < old_path->rows)
             remove_old = true;    /* new dominates old */
          //如果新路径返回的行数多，选择不接受新路径，保留旧路径
          else if (new_path->rows > old_path->rows)
             accept_new = false; /* old dominates new */
         //到这里，代价、PathKeys、路径参数、并行安全、结果集行数均相似
         //“收紧”代价判断的范围，如果新路径好，则采用新路径，放弃旧路径
          else if (compare_path_costs_fuzzily(new_path,
                                    old_path,
                                   1.0000000001) == COSTS_BETTER1)
             remove_old = true;    /* new dominates old */
          else
             accept_new = false; /* old equals or
                                  * dominates new */
       }
       else if (outercmp == BMS_SUBSET1 &&
             new_path->rows <= old_path->rows &&
             new_path->parallel_safe >= old_path->parallel_safe)
          //如果代价和 Pahtkeys 相似，其他方面新路径均好于或等于旧路径
          //则放弃旧路径，保留新路径
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
                remove_old = true;    /* new dominates old */
            else if (outercmp == BMS_SUBSET2 &&
                     new_path->rows >= old_path->rows &&
                     new_path->parallel_safe <= old_path->parallel_safe)
                //如果代价和 Pahtkeys 相等或相似，其他方面旧路径均差于或等于新路径
                //则不接受新路径，保留旧路径
                accept_new = false; /* old dominates new */
            /* else different parameterizations, keep both */
        }
        break;
    case COSTS_BETTER1:
        //所有判断因素新路径均好于或等于旧路径，则接受新路径，放弃旧路径
        if (keyscmp != PATHKEYS_BETTER2)
        {
            outercmp = bms_subset_compare(PATH_REQ_OUTER(new_path),
                                          PATH_REQ_OUTER(old_path));
            if ((outercmp == BMS_EQUAL ||
                 outercmp == BMS_SUBSET1) &&
                new_path->rows <= old_path->rows &&
                new_path->parallel_safe >= old_path->parallel_safe)
                remove_old = true;    /* new dominates old */
        }
        break;
    case COSTS_BETTER2:
        //所有判断因素旧路径均差于或等于新路径，则不接受新路径，保留旧路径
        if (keyscmp != PATHKEYS_BETTER1)
        {
            outercmp = bms_subset_compare(PATH_REQ_OUTER(new_path),
                                          PATH_REQ_OUTER(old_path));
            if ((outercmp == BMS_EQUAL ||
                 outercmp == BMS_SUBSET2) &&
                new_path->rows >= old_path->rows &&
                new_path->parallel_safe <= old_path->parallel_safe)
                accept_new = false; /* old dominates new */
        }
        break;
    case COSTS_DIFFERENT:
        break;
    }
  }
}
```


非卖品，仅供非商业用途或交流学习使用

8.7 小结

本章关于连接合法性判断的部分是比较难以理解的，其中对有些情况的判断 PostgreSQL 数据库开发人员也没有办法给出示例，读者想要理解其中的含义需要仔细分析这部分的源代码。

本章重点介绍了物理连接路径的生成，实际上物理连接路径的种类比较单一，只有 Nestlooped Join、Hash Join、MergeJoin 这么 3 种情况，不过结合不同类型的扫描路径，例如并行扫描路径、参数化扫描路径、索引扫描路径等，就会产生很多不同的可能性。

非卖品，仅供非商业用途或交流学习使用

9

第 9 章
Non-SPJ 优化

grouping_planner 函数中主要处理了一些 Non-SPJ 的优化，其中主要包括集合操作、聚集操作、Limit 子句处理、Order By 子句处理、Group By 子句处理等。

在之前的章节中已经介绍过基于 Limit 子句产生的 tuple_fraction 变量和基于其他子句产生的 PathKeys 对下层的 SPJ 操作产生路径有提示的作用，本章则主要介绍如下两个方面。

- 集合操作的实现流程：主要关注集合操作“本身”的流程，因为集合操作的实现到了下层是子查询的 SPJ 优化，这部分已经介绍过了。
- 在 SPJ 优化路径上叠加 Non-SPJ 路径的过程：主要介绍基于不同 Non-SPJ 子句的特性做一些特殊处理。

9.1 集合操作处理

集合操作包含 UNION [ALL]、INTERSECT [ALL]、EXCEPT [ALL]，如果不带有 ALL 关键字，则有去重的语义包含在里面，例如有两个表 TEST_A 和 TEST_B，它们的数据如下：

```
postgres=# SELECT * FROM TEST_A;
 a | b | c | d
---+---+---+---
 1 | 2 | 3 | 4
 2 | 2 | 3 | 4
 1 | 2 | 3 | 4
 2 | 2 | 3 | 4
(4 rows)
postgres=# SELECT * FROM TEST_B;
 a | b | c | d
---+---+---+---
 1 | 2 | 3 | 4
 3 | 2 | 3 | 4
 1 | 2 | 3 | 4
(3 rows)
```

UNION ALL 操作和 UNION 操作的区别如下：

```
postgres=# SELECT * FROM TEST_A UNION ALL SELECT * FROM TEST_B;
 a | b | c | d
---+---+---+---
 1 | 2 | 3 | 4
 2 | 2 | 3 | 4
 1 | 2 | 3 | 4
 2 | 2 | 3 | 4
 1 | 2 | 3 | 4
 3 | 2 | 3 | 4
 1 | 2 | 3 | 4
(7 rows)
postgres=# SELECT * FROM TEST_A UNION SELECT * FROM TEST_B;
 a | b | c | d
---+---+---+---
 3 | 2 | 3 | 4
 1 | 2 | 3 | 4
 2 | 2 | 3 | 4
(3 rows)
```

INTERSECT ALL 操作和 INTERSECT 操作的区别如下：

```
postgres=# SELECT * FROM TEST_A INTERSECT ALL SELECT * FROM TEST_B;
 a | b | c | d
---+---+---+---
 1 | 2 | 3 | 4
```

```
 1 | 2 | 3 | 4
(2 rows)
postgres=# SELECT * FROM TEST_A INTERSECT SELECT * FROM TEST_B;
 a | b | c | d
---+---+---+---
 1 | 2 | 3 | 4
(1 row)
```

EXCEPT ALL 操作和 EXCEPT 操作的区别如下：

```
postgres=# SELECT * FROM TEST_A EXCEPT ALL SELECT * FROM TEST_B;
 a | b | c | d
---+---+---+---
 2 | 2 | 3 | 4
 2 | 2 | 3 | 4
(2 rows)
postgres=# SELECT * FROM TEST_A EXCEPT SELECT * FROM TEST_B;
 a | b | c | d
---+---+---+---
 2 | 2 | 3 | 4
(1 row)
```

在集合操作的外层，PostgreSQL 数据库还支持对集合操作的结果进行 Order By 操作、Locking 操作和 Limit 操作，例如：

```
postgres=# SELECT * FROM TEST_A UNION SELECT * FROM TEST_B ORDER BY a,b,c LIMIT 2;
 a | b | c | d
---+---+---+---
 1 | 2 | 3 | 4
 2 | 2 | 3 | 4
(2 rows)
```

其中 Order By 子句先于 Limit 子句执行，从语法定义中可以看出。

```
select_no_parens:
        simple_select                    { $$ = $1; }
        | select_clause sort_clause
        | select_clause opt_sort_clause for_locking_clause opt_select_limit
        | select_clause opt_sort_clause select_limit opt_for_locking_clause
select_clause:
        simple_select                      { $$ = $1; }
        | select_with_parens               { $$ = $1; }
simple_select:
```

```
        | select_clause UNION all_or_distinct select_clause
        | select_clause INTERSECT all_or_distinct select_clause
        | select_clause EXCEPT all_or_distinct select_clause
```

由于 Limit 子句需要在 Order By 的基础上进行，而 Order By 操作则需要对所有集合操作的结果进行排序，这时即使基于 Limit 子句获得了 tuple_fraction 变量也没什么用，因此集合操作路径生成过程中如果发现有 Order By 子句，就忽略 tuple_fraction 的值。

```
//如果集合操作的结果需要先做 ORDER BY，然后做 Limit/offset
//由于 Order By 需要对集合操作的全部结果元组进行排序
//那么忽略 Limit 子句对 tuple_fraction 的影响，tuple_fraction=0.0 表示获取全部元组
if (parse->sortClause)
   root->tuple_fraction = 0.0;
```

集合操作本质上是多个子查询之间进行交并差的操作，因此集合操作生成的执行计划中，叶子节点是子查询路径，例如：

```
postgres=# EXPLAIN VERBOSE SELECT * FROM TEST_A UNION SELECT * FROM TEST_B ORDER BY a,b,c
LIMIT 2;
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Limit  (cost=205.00..205.01 rows=2 width=16)
   Output: test_a.a, test_a.b, test_a.c, test_a.d
   -> Sort  (cost=205.00..214.25 rows=3700 width=16)
        Output: test_a.a, test_a.b, test_a.c, test_a.d
        Sort Key: test_a.a, test_a.b, test_a.c
        -> HashAggregate  (cost=131.00..168.00 rows=3700 width=16)
             Output: test_a.a, test_a.b, test_a.c, test_a.d
             Group Key: test_a.a, test_a.b, test_a.c, test_a.d
             -> Append  (cost=0.00..94.00 rows=3700 width=16)
                  -> Seq Scan on public.test_a (cost=0.00..28.50 rows=1850 width=16)
                       Output: test_a.a, test_a.b, test_a.c, test_a.d
                  -> Seq Scan on public.test_b (cost=0.00..28.50 rows=1850 width=16)
                       Output: test_b.a, test_b.b, test_b.c, test_b.d
(13 rows)
```

示例 SQL 的执行计划中叶子节点是两个 SeqScan 扫描路径（本质上是 SubqueryScan->SeqScan，SubqueryScan 在执行计划生成时被优化掉了），但是到了上层的投影仍然显示的是 TEST_A 的投影（test_a.a, test_a.b, test_a.c, test_a.d），这是因为集合操作借用了 TEST_A 的子查询的列名作为投影属性名，PostgreSQL 数据库默认借用最左端叶子节点的列名，示例中 TEST_A 是最左端的叶子节点。

```
//使用最左端的叶子节点
node = topop->larg;
while (node && IsA(node, SetOperationStmt))
   node = ((SetOperationStmt *) node)->larg;
Assert(node && IsA(node, RangeTblRef));
leftmostRTE = root->simple_rte_array[((RangeTblRef *) node)->rtindex];
leftmostQuery = leftmostRTE->subquery;
```

集合操作的投影列是根据最左端的叶子节点的列名和集合操作本身记录的要投影列的类型（SetOperationStmt->colTypes）合并而成的，创建投影列分别使用了 generate_setop_tlist 函数和 generate_append_tlist 函数，集合操作的执行计划本质上还是通过 Append 的方式来实现的，generate_setop_tlist 函数负责对执行计划的叶子节点上的路径生成投影列，它是有具体的基表的，因此有对应的 rtindex，generate_append_tlist 函数则是基于几个子查询节点进行集合操作之后产生的新的投影列，也就是说它用来描述集合操作的中间结果的投影，它的 rtindex 对应的值是 0。

```
//有 rtindex 的情况——generate_setop_tlist 函数
expr = (Node *) makeVar(varno,
                  inputtle->resno,
                  exprType((Node *) inputtle->expr),
                  exprTypmod((Node *) inputtle->expr),
                  exprCollation((Node *) inputtle->expr),
                  0);
//rtindex 是 0 的情况——generate_append_tlist 函数
expr = (Node *) makeVar(0,
                  resno,
                  colType,
                  colTypmod,
                  colColl,
                  0);
```

在集合操作中，子节点的数据类型不一定一致，这时候就需要对数据类型进行隐式的强制转换。强制转换在叶子节点上进行，也就是在 generate_setop_tlist 函数中进行，这样能尽早地在下层对数据进行转换，上层就可以放心地使用了，例如：

```
CREATE TABLE TEST_INT2(a INT2);
CREATE TABLE TEST_INT4(a INT4);
CREATE TABLE TEST_INT8(a INT8);
postgres=# EXPLAIN SELECT * FROM TEST_INT2 UNION ALL SELECT * FROM TEST_INT4 UNION ALL SELECT
* FROM TEST_INT8;
                              QUERY PLAN
-------------------------------------------------------------------------------------
```

```
Append  (cost=0.00..253.27 rows=7530 width=8)
  ->  Result  (cost=0.00..198.07 rows=5270 width=8)
        ->  Append  (cost=0.00..132.20 rows=5270 width=4)
              ->  Subquery Scan on "*SELECT* 1"  (cost=0.00..71.20 rows=2720 width=4)
                    ->  Seq Scan on test_int2  (cost=0.00..37.20 rows=2720 width=2)
              ->  Seq Scan on test_int4  (cost=0.00..35.50 rows=2550 width=4)
  ->  Seq Scan on test_int8  (cost=0.00..32.60 rows=2260 width=8)
(7 rows)
```

在 TEST_INT2 UNION ALL TEST_INT4 中需要将 INT2 类型的数据转换成 INT4 类型的数据（从执行计划可以看出，TEST_INT2 表的扫描路径中没有优化掉 SubqueryScan，就是由于 SeqScan 的投影列和 SubqueryScan 的投影列不同，SeqScan 的投影列中都是简单的 Var，而 SubqueryScan 的投影列中需要将 INT2 转换成 INT4，是类型转换的表达式），TEST_INT2 UNION ALL TEST_INT4 的结果和 TEST_INT8 做 UNION ALL 时，会将 INT4 转换成 INT8。

集合操作的路径生成区分成了 3 种情况，RecursiveUnion、UNION、Non-UNION(EXCEPT& INTERSECT)，如图 9-1 所示。

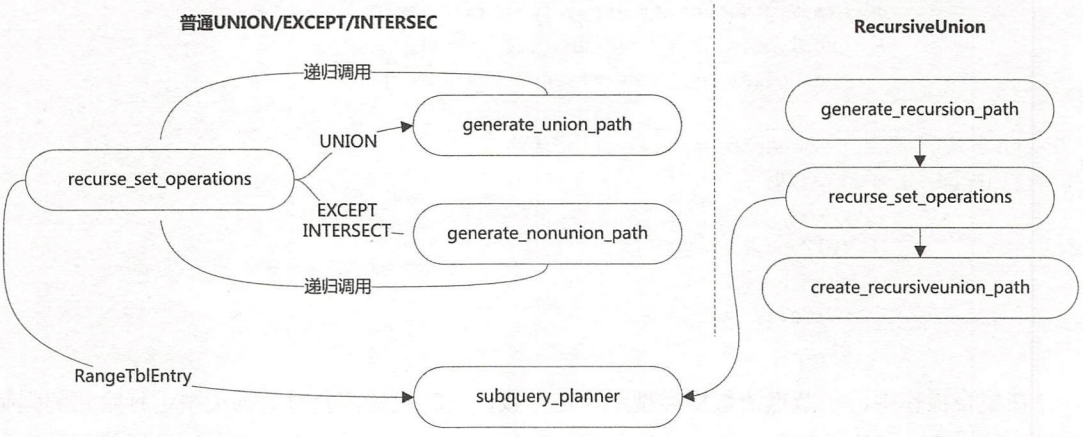


图 9-1　集合操作函数调用关系

以 RecursiveUnion 路径为例，它的执行计划如下：

```
postgres=# EXPLAIN WITH RECURSIVE CTE AS
postgres-# (
postgres-#    SELECT a.a FROM TEST_A a WHERE a =1 UNION ALL
postgres-#    SELECT b.a FROM TEST_A b INNER JOIN CTE c on b.a = c.a
postgres-# )SELECT a FROM CTE;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------------
 CTE Scan on cte  (cost=2085.47..2087.49 rows=101 width=4)
   CTE cte
     -> Recursive Union  (cost=0.00..2085.47 rows=101 width=4)
          -> Seq Scan on test_a a  (cost=0.00..177.85 rows=1 width=4)
               Filter: (a = 1)
          -> Hash Join  (cost=0.33..190.56 rows=10 width=4)
               Hash Cond: (b.a = c.a)
               -> Seq Scan on test_a b  (cost=0.00..153.28 rows=9828 width=4)
               -> Hash  (cost=0.20..0.20 rows=10 width=4)
                    -> WorkTable Scan on cte c  (cost=0.00..0.20 rows=10 width=4)
(10 rows)
```

它们的子查询最终都会进入 subquery_planner 函数做逻辑优化和物理优化，这里不再赘述。

9.2　Non-SPJ 路径

在 9.6 版本之前，Non-SPJ 操作是直接生成执行计划（Plan）的，每个 Non-SPJ 操作对应一个 Plan 节点，create_plan 函数主要针对 SPJ 操作产生的路径进行操作，由于 create_plan 函数针对的是某一条路径（通常是最优路径），就需要快速地在 SPJ 优化之后生成的路径中选出一个最优的路径来生成 SPJ 路径对应的 Plan，然后再将 Non-SPJ 操作对应的 Plan 节点叠加到 SPJ 的 Plan 上（更早之前的版本在 SPJ 优化阶段直接生成 Plan，到 grouping_planner 函数中直接叠加 Non-SPJ 的 Plan 节点）。

在 9.6 版本之后，query_planner 函数直接返回 SPJ 优化之后产生的最顶端的 RelOptInfo，这个 RelOptInfo 中记录了 SPJ 优化后最顶端的各种路径，这些路径都可以被用到，返回的路径多了获取“最”优路径可能性就大了，当然生成执行计划的时间也会变多。

在 grouping_planner 函数中，以 SPJ 优化函数 query_planner 为分水岭。

- 在 query_planner 函数之前，对 Non-SPJ 操作中的各种表达式进行预处理。
- 在 query_planner 函数中，是 SPJ 优化路径的生成。
- 在 query_planner 函数之后，开始在 query_planner 函数生成的 SPJ 优化路径上叠加新的 Non-SPJ 路径。

9.2.1 Non-SPJ 预处理

Non-SPJ 的预处理包括对 Group By 子句、聚集函数的代价、投影列等的预处理，这部分一方面是对一些能够优化的部分进行优化，例如在处理 Group By 子句的过程中多次尝试了是否可以共享 Order By 子句的排序，另一方面是为后续创建路径的工作做准备，例如在聚集函数预处理时为将来创建 AggPath 的代价计算做准备等。

9.2.1.1 Group By 子句的预处理

Group By 子句中可能包含 GROUPING SET、ROLLUP、CUBE，这种情况需要做特殊的预处理，如表 9-1 所示。

表 9-1 Group By中的等价关系

Group By类型	等价的表达式
GROUP BY a,b GROUPING SETS{c,d}	GROUP BY a,b,c GROUP BY a,b,d
GROUP BY a,b, ROLLUP(c, d)	GROUP BY a,b,c,d GROUP BY a,b,c GROUP BY a,b
GROUP BY a,b, CUBE(c, d)	GROUP BY a,b,c,d GROUP BY a,b,c GROUP BY a,b GROUP BY a,b,d

例如：

```
postgres=# EXPLAIN SELECT * FROM TEST_A GROUP BY a, CUBE(b, c, d);
                        QUERY PLAN
-----------------------------------------------------------------------
 HashAggregate  (cost=251.56..1430.92 rows=78624 width=16)
   Hash Key: a, b, c, d
   Hash Key: a, b, c
   Hash Key: a, b
   Hash Key: a
   Hash Key: a, c, d
   Hash Key: a, c
   Hash Key: a, d, b
   Hash Key: a, d
   ->  Seq Scan on test_a  (cost=0.00..153.28 rows=9828 width=16)
(10 rows)
```

从示例中可以看出，由于 CUBE(b, c, d)的语义是：

{b} ∪ {c} ∪ {d} ∪ {b,c} ∪ {b,d} ∪ {c,d} ∪ {b,c,d} ∪ {}

共 8 种情况，这 8 种情况和{a}的卡氏积就是示例中等价的 8 种情况，expand_grouping_sets 函数的主要作用是对包含 GROUPING SETS、ROLLUP、CUBE 的子句进行扩展，生成它们的卡氏积，然后对卡氏积的结果按照从短到长的顺序排序。

```
//以 GROUP BY a, CUBE(b, c, d)为例
List * expand_grouping_sets(List *groupingSets, int limit)
{
   //扩展：例如 CUBE(b,c,d)这样的表达式需要转换成：
   //{b} ∪ {c} ∪ {d} ∪ {b,c} ∪ {b,d} ∪ {c,d} ∪ {b,c,d} ∪ {}
   foreach(lc, groupingSets)
   {
      current_result = expand_groupingset_node(gs);
      //计算卡氏积结果的大小 = 1×8 = 8
      numsets *= list_length(current_result);
   }

   //做卡氏积，生成等价的情况
   //{a,b} ∪ {a,c} ∪ {a,d} ∪ {a,b,c} ∪ {a,b,d} ∪ {a,c,d} ∪ {a,b,c,d} ∪ {a}
   for_each_cell(lc, lnext(list_head(expanded_groups)))
      ......
   //排序：从短到长的顺序排列
   //{a} ∪ {a,b} ∪ {a,c} ∪ {a,d} ∪ {a,b,c} ∪ {a,b,d} ∪ {a,c,d} ∪ {a,b,c,d}
   if (list_length(result) > 1)
      ......
   return result;
}
```

仔细观察前面的示例不难发现，示例中的各个 Hash Key 并不是按照从短到长的顺序进行排列的，这里进行排序的原因是为后面的优化做准备，对于多个 Group Key，如果通过排序的方法进行分组，有些 Group Key 是能共用同一次排序结果的，如下示例中展示了 Sort 聚集的执行计划。

```
postgres=# set enable_hashagg=false;
SET
postgres=# EXPLAIN SELECT * FROM TEST_A GROUP BY a, CUBE(b,c,d);
                        QUERY PLAN
------------------------------------------------------------------------
```

```
GroupAggregate  (cost=805.01..3214.11 rows=78624 width=16)
  Group Key: a, b, c, d
  Group Key: a, b, c
  Group Key: a, b
  Group Key: a
  Sort Key: a, c, d
    Group Key: a, c, d
    Group Key: a, c
  Sort Key: a, d, b
    Group Key: a, d, b
    Group Key: a, d
  ->  Sort  (cost=805.01..829.58 rows=9828 width=16)
        Sort Key: a, b, c, d
        ->  Seq Scan on test_a  (cost=0.00..153.28 rows=9828 width=16)
(14 rows)
```

从示例可以看出：{a,b,c,d}、{a,b,c}、{a,b}、{a}使用的是基于排序键 a,b,c,d 的排序结果，{a,c,d}、{a,c}使用的是基于排序键 a,c,d 的排序结果，{a,d,b}、{a,d}使用的是基于排序键 a,d,b 的排序结果，再仔细观察则可以反推出上面的 3 个分类实际上是 3 个基于 ROLLUP 语义的 Group Key（需要去掉重复的 Group Key）。

$$\begin{aligned}
\text{ROLLUP(a,b,c,d)} &= \{a,b,c,d\} \cup \{a,b,c\} \cup \{a,b\} \cup \{a\} \\
\text{ROLLUP(a,c,d)} &= \{a,c,d\} \cup \{a,c\} \\
\text{ROLLUP(a,b,d)} &= \{a,d,b\} \cup \{a,d\}
\end{aligned}$$

因此可以考虑将原来的 8 个 Group Key 划分成 3 组 ROLLUP 的方式来求解，这部分是在 extract_rollup_sets 函数中实现的，划分的过程采用的是 Hopcroft–Karp 算法，Hopcroft–Karp 算法是一种二分图的最大匹配算法，我们以 GROUP BY a, CUBE(b,c,d)为例来看一下形成 3 组 ROLLUP 的过程。

如图 9-2 所示是一个二分图，它的左侧 U 为 8 种情况，它的右侧 V 是左侧 U 对应的真子集，例如{a,b,c}的真子集可以是{a}、{a,b}、{a,c}，连线是子集之间的关联情况，U 和 V 构成了一个二分图，加粗的连线是二分图的最大匹配。

最大匹配的连线关系是{2, 1}、{5,2}、{7,3}、{6,4}、{8,5}，其中{2,1}代表 U 中的 1 和 V 中的 2 是最大匹配的一条边，基于最大匹配的连线关系，可以形成 3 个分组，如表 9-1 所示和图 9-2 所示。

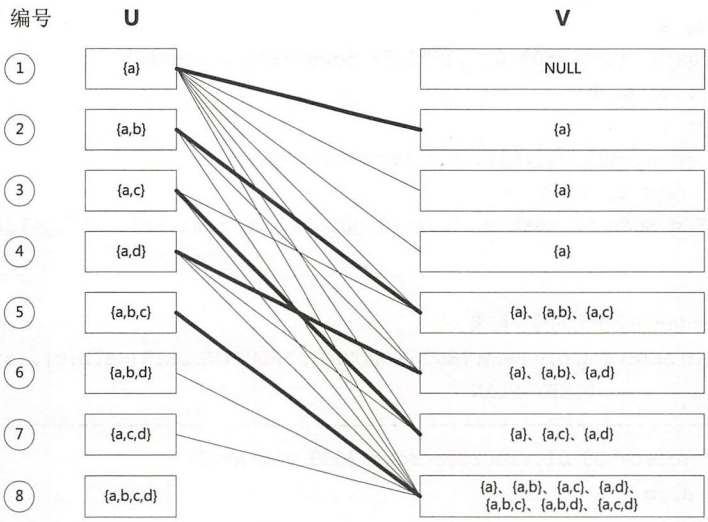


图 9-2　二分图的最大匹配

```
//chains 能够形成一个链，里面以标号的形式形成分组
//以 GROUP BY a,CUBE(b,c,d)为例，形成的 chains 为:
//{1,1,2,3,1,3,2,1}
for (i = 1; i <= num_sets; ++i)
{
   int u = state->pair_vu[i];
   int v = state->pair_uv[i];

   if (u > 0 && u < i)
      chains[i] = chains[u];
   else if (v > 0 && v < i)
      chains[i] = chains[v];
   else
      chains[i] = ++num_chains;
}
```

图 9-3 中对于同一个 ROLLUP 中的 Group Key 还进行了排序，在排序的同时还会判断是否可以借用 Order By 子句的排序，这部分在 reorder_grouping_sets 函数中完成，例如：

```
-- 示例 1：没有借用 Order by 子句的排序结果
postgres=# EXPLAIN SELECT a,b,c FROM TEST_A GROUP BY GROUPING SETS((a,b,c),(c)) ORDER BY c,b,a;
                              QUERY PLAN
------------------------------------------------------------------------------
 Sort  (cost=1661.74..1686.56 rows=9929 width=12)
```

```
  Sort Key: c, b, a
  -> GroupAggregate  (cost=805.01..1002.58 rows=9929 width=12)
       Group Key: c, a, b
       Group Key: c
       -> Sort  (cost=805.01..829.58 rows=9828 width=12)
            Sort Key: c, a, b
            -> Seq Scan on test_a  (cost=0.00..153.28 rows=9828 width=12)
(8 rows)

-- 示例 2：借用了 Order By 子句的排序结果
postgres=# EXPLAIN SELECT a,b,c FROM TEST_A GROUP BY GROUPING SETS((a,b,c),(c)) ORDER BY c,a,b;
                        QUERY PLAN
-----------------------------------------------------------------------------
 GroupAggregate  (cost=805.01..1002.58 rows=9929 width=12)
   Group Key: c, a, b
   Group Key: c
   -> Sort  (cost=805.01..829.58 rows=9828 width=12)
        Sort Key: c, a, b
        -> Seq Scan on test_a  (cost=0.00..153.28 rows=9828 width=12)
(6 rows)
```

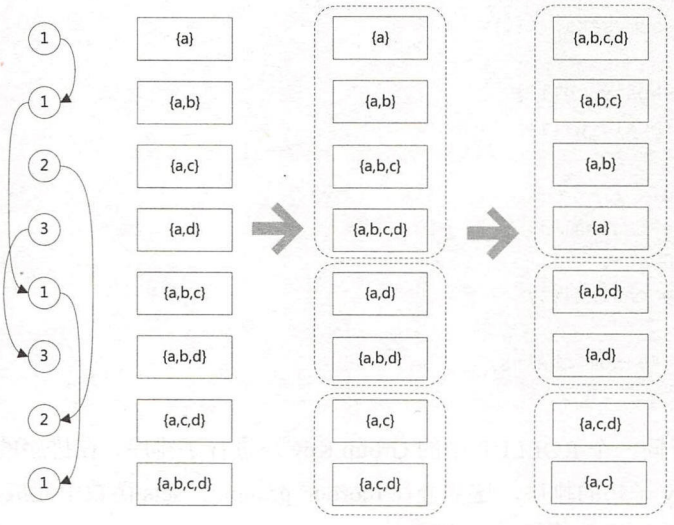


图 9-3　通过排序进行分组

虽然示例 1 和示例 2 的结果应该是相同的（Group Keys 顺序交换，不影响查询结果），但是一个借用了 Order By 子句的排序结果，别一个没有借用 Order By 子句的排序结果，也就是说

目前的查询优化器没有足够的“智慧”对示例 1 进行优化。

如果 Group By 子句中不包括 GROUPING SETS、ROLLUP、CUBE 这几种情况，只是普通的 Group By 子句，也需要考虑借用 Order By 的结果（preprocess_groupclause 函数），例如：

```
postgres=# EXPLAIN SELECT a,b,c FROM TEST_A GROUP BY a,b,c ORDER BY b,c,a;
                        QUERY PLAN
------------------------------------------------------------------------
 Group  (cost=805.01..903.29 rows=9828 width=12)
   Group Key: b, c, a
   ->  Sort  (cost=805.01..829.58 rows=9828 width=12)
         Sort Key: b, c, a
         ->  Seq Scan on test_a  (cost=0.00..153.28 rows=9828 width=12)
(5 rows)
```

由于 Group By 子句中的分组键之间顺序交换之后不影响查询语句的结果，因此示例中的 GROUP BY a,b,c 和 ORDER BY b,c,a 是能够匹配上的。

9.2.1.2　投影列的预处理

目前投影列主要保存在 Query-> targetList 中，对投影列的预处理没有集中在查询语句上，它主要针对如下情况：

- 插入（INSERT）和（UPDATE）操作的投影列需要补齐。
- SELECT…FOR UPDATE 需要在投影列中增加伪列。
- RETURNING 子句处理返回的投影列。
- 预处理 ON CONFLICT 的投影列。

以上这些主要集中在插入（INSERT）和更新（UPDATE）操作上，和查询优化的关系不大，本书不做重点介绍。

9.2.1.3　聚集函数预处理

对查询树（Query）进行遍历，获得其中的聚集函数节点，聚集函数主要出现在投影列和 Having 子句中。

```
//遍历投影列中的聚集函数
get_agg_clause_costs(root, (Node *) tlist, AGGSPLIT_SIMPLE, &agg_costs);
//遍历 Having 子句中的聚集函数
get_agg_clause_costs(root, parse->havingQual, AGGSPLIT_SIMPLE, &agg_costs);
```

在计算代价之前，还会统计聚集函数中的一些信息，例如聚集函数是否支持并行。

```
//aggref->aggorder 示例:
//       SELECT mode() WITHIN GROUP (ORDER BY b) FROM TEST_A GROUP BY a
//aggref->aggdistinct 示例: SELECT SUM(DISTINCT a) FROM TEST_A
if (aggref->aggorder != NIL || aggref->aggdistinct != NIL)
{
   //costs->numOrderedAggs 负责统计查询中有几个聚集函数包含 DISTINCT/WITHIN GROUP
   costs->numOrderedAggs++;

   //如果查询中有聚集函数包含 DISTINCT/WITHIN GROUP，则不能创建并行聚集路径
   costs->hasNonPartial = true;
}

if (!costs->hasNonPartial)
{
   //并行聚集必须需要 combine 函数，否则不能创建并行路径，
   //聚集相关函数可参考 PG_AGGREGATE 系统表
   if (!OidIsValid(aggcombinefn))
      costs->hasNonPartial = true;

   //聚集函数的中间类型是 INTERNAL 类型，如果要实现并行聚集，
   //必须有序列化和反序列化函数，聚集相关函数可参考 PG_AGGREGATE 系统表
   else if (aggtranstype == INTERNALOID &&
         (!OidIsValid(aggserialfn) || !OidIsValid(aggdeserialfn)))
      costs->hasNonSerial = true;
}
```

同时针对聚集函数衍生出的各种操作函数计算其表达式代价及估计聚集函数的数据 transition 需要的空间。

```
//计算聚集函数衍生的各种操作符的代价
if (DO_AGGSPLIT_COMBINE(context->aggsplit))
{
   /* charge for combining previously aggregated states */
   costs->transCost.per_tuple += get_func_cost(aggcombinefn) * cpu_operator_cost;
}
else
   costs->transCost.per_tuple += get_func_cost(aggtransfn) * cpu_operator_cost;
if (DO_AGGSPLIT_DESERIALIZE(context->aggsplit) && OidIsValid(aggdeserialfn))
   costs->transCost.per_tuple += get_func_cost(aggdeserialfn) * cpu_operator_cost;
if (DO_AGGSPLIT_SERIALIZE(context->aggsplit) && OidIsValid(aggserialfn))
   costs->finalCost += get_func_cost(aggserialfn) * cpu_operator_cost;
if (!DO_AGGSPLIT_SKIPFINAL(context->aggsplit) && OidIsValid(aggfinalfn))
```

```
    costs->finalCost += get_func_cost(aggfinalfn) * cpu_operator_cost;
......

//值传递的变量不计算 transition 空间
if (!get_typbyval(aggtranstype))
    ......
```

9.2.1.4　MIN/MAX 优化预处理

因为 B 树索引是有序的，如果要获取某一列的最大值或者最小值，同时这一列是 B 树索引的键值，那么可以借用索引来获取最大值，例如：

```
postgres=# CREATE INDEX TEST_A_A_IDX ON TEST_A(a);
postgres=# CREATE INDEX TEST_A_B_IDX ON TEST_A(b);
postgres=# EXPLAIN SELECT MAX(a), MIN(b) FROM TEST_A WHERE c > 1;
                                  QUERY PLAN
-------------------------------------------------------------------------------------
 Result  (cost=0.66..0.67 rows=1 width=8)
   InitPlan 1 (returns $0)
     -> Limit  (cost=0.29..0.32 rows=1 width=4)
          -> Index Scan Backward using test_a_a_idx on test_a
               Index Cond: (a IS NOT NULL)
               Filter: (c > 1)
   InitPlan 2 (returns $1)
     -> Limit  (cost=0.29..0.34 rows=1 width=4)
          -> Index Scan using test_a_b_idx on test_a test_a_1
               Index Cond: (b IS NOT NULL)
               Filter: (c > 1)
(11 rows)
```

从执行计划可以看出，针对每个 MAX/MIN 聚集函数增加了一个对应的子 Plan，这个子 Plan 是在 build_minmax_path 函数中创建的，子 Plan 是基于如下 SQL 语句生成的：

```
SELECT col FROM tab
   WHERE col IS NOT NULL AND existing-quals
   ORDER BY col ASC/DESC
   LIMIT 1;
```

判断 MIN/MAX 优化能否适用的条件是比较严格的，比较主要的要求如下：

- 只能包含 MIN/MAX 聚集函数，语句中不能有其他聚集函数。
- 如果是多个表连接的情况，不适用 MIM/MAX 优化。

- 查询中不能包含 Group By 子句，因为创建了分组之后就无法借用索引获得 MIN/MAX 值。
- 语句中不能包含 CTE 表达式。
- 如果 MIN/MAX 对应的列上没有 B 树索引，也不适用这种优化方式。

9.2.2 Non-SPJ 路径生成

在 query_planner 函数中生成了基于 SPJ 操作（以 Scan/Join 为基础）的路径树，Non-SPJ 路径是在 SPJ 路径的基础上进行叠加，例如：

```
postgres=# EXPLAIN SELECT a,b,c,sum(d) as d FROM TEST_A GROUP BY a,b,c ORDER BY d LIMIT 10;
                              QUERY PLAN
-----------------------------------------------------------------------------------------
 Limit  (cost=1260.48..1260.51 rows=10 width=20)
   ->  Sort  (cost=1260.48..1285.48 rows=10000 width=20)
         Sort Key: (sum(d))
         ->  GroupAggregate  (cost=819.39..1044.39 rows=10000 width=20)
               Group Key: a, b, c
               ->  Sort  (cost=819.39..844.39 rows=10000 width=16)
                     Sort Key: a, b, c
                     ->  Seq Scan on test_a  (cost=0.00..155.00 rows=10000 width=16)
(8 rows)
```

示例中的 SPJ 部分生成的路径是一个 SeqScan 路径，在 SeqScan 的基础上依次叠加了分组聚集（基于排序的分组，实现 Group By 子句）、排序（实现 Order By 子句）、Limit 操作（实现 Limit 子句）。

每一种 Non-SPJ 生成的路径都记录到对应的 RelOptInfo 中，PostgreSQL 生成了对应的数组，数组下标的含义如下：

```
typedef enum UpperRelationKind
{
    UPPERREL_SETOP,             //集合操作的结果路径
    UPPERREL_GROUP_AGG,         //Group By 子句和聚集函数对应的路径
    UPPERREL_WINDOW,            //窗口函数对应的路径
    UPPERREL_DISTINCT,          //去重操作对应的路径
    UPPERREL_ORDERED,           //Order By 对应的路径
    UPPERREL_FINAL              //叠加了 Non-SPJ 之后的路径
    /* NB: UPPERREL_FINAL must be last enum entry; it's used to size arrays */
} UpperRelationKind;
```

9.2.2.1　投影处理

如果投影中有 Order By 子句，投影中的一些易失性表达式可能因为路径的不同而产生不同的投影结果，例如：

```
CREATE TEMP SEQUENCE SEQ;
SELECT a, nextval('SEQ') FROM TEST_A ORDER BY a LIMIT 10;
```

它可能产生的执行计划如下：

路径 1：显式地排序

```
LIMIT ( Project: a, res )
     ->Sort ( Project: a, nextval('SEQ') as res )
               ->SeqScan ( Project: a )
```

路径 2：借用了 IndexScan 的有序特性（假设索引已经建立好了）

```
LIMIT ( Project: a, nextval('SEQ') as res )
               ->IndexScan ( Project: a )
```

路径 1 和路径 2 产生的结果是不同的，路径 1 对 TEST_A 表中的所有行都获取了一次 nextval('SEQ')，而路径 2 只进行了 10 次 nextval('SEQ')，这会带来 3 个问题。

- 路径 1 执行 nextval('SEQ')的次数多，显而易见代价高。
- 两条路径执行之后序列（Sequence）的实际的 nextval('SEQ')不一致，路径 1 执行之后的 nextval('SEQ')显然和路径 2 执行之后的 nextval('SEQ')不同。
- 路径 1 和路径 2 的执行结果不同，路径 1 中的 nextval('SEQ')投影的结果会先于排序产生，在排序的过程中会被打乱，因此这一列对应的最终的结果不是有序的，而路径 2 借用了索引有序的特点，没有打乱 nextval('SEQ')的顺序，因此它显示的结果是有序的。

如果要使路径 1 和路径 2 产生的结果相同，可以对路径 1 进行改造，将其改造成如下形式：

路径 1 改进：显式地排序

```
LIMIT ( Project: a, nextval('SEQ') as res)
     ->Sort ( Project: a )
               ->SeqScan ( Project: a )
```

我们对比一下实际的执行计划可以看出，PostgreSQL 数据库将 nextval('SEQ')转移到了排序之后执行。

```
postgres=# EXPLAIN VERBOSE SELECT a, nextval('SEQ') FROM TEST_A ORDER BY a LIMIT 10;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Limit  (cost=371.10..371.25 rows=10 width=12)
   Output: a, (nextval('seq'::regclass))
```

```
  ->  Result  (cost=371.10..521.10 rows=10000 width=12)
        Output: a, nextval('seq'::regclass)
        ->  Sort  (cost=371.10..396.10 rows=10000 width=4)
              Output: a
              Sort Key: test_a.a
              ->  Seq Scan on public.test_a  (cost=0.00..155.00 rows=10000 width=4)
                    Output: a
(9 rows)
```

这种将易失性表达式的求值保留在高层节点的方法还可以引申出来一些性能上的优化，在make_sort_input_target 函数中，它处理了 3 种优化情况。

- 易失性表达式需要在高层节点计算。
- SRF 表达式需要在高层节点计算，例如 generate_series 函数。
- 投影中如有代价比较高的表达式（> 10 × cpu_operator_cost），同时查询语句中有 Limit 子句，这种表达式需要在高层节点计算。

另外，由于 query_planner 函数只做了基于 SPJ 的优化，因此它产生的路径的投影（Path->pathtarget）不一定是查询最上层的投影，比如在有聚集函数的情况下，查询的最上层的投影需要显示的是聚集函数的结果，但是 SPJ 优化产生的路径无法产生这种聚集函数的投影。

```
postgres=# EXPLAIN VERBOSE SELECT SUM(A) FROM TEST_A;
                          QUERY PLAN
------------------------------------------------------------------------
 Aggregate  (cost=180.00..180.01 rows=1 width=8)
   Output: sum(a)
   ->  Seq Scan on public.test_a  (cost=0.00..155.00 rows=10000 width=4)
         Output: a, b, c, d
(4 rows)
```

从示例中可以看出，基于 SPJ 的优化生成的是 SeqScan 路径，它的投影是 TEST_A 表的 4 个列（注：在生成 Path 阶段只有 a 列，在 create_plan 阶段优化为对 TEST_A 表全部的列投影），而在 Non-SPJ 阶段的投影是 sum(a)，SPJ 阶段的投影和 Non-SPJ 阶段的投影是不同的，因此需要根据 Non-SPJ 阶段的投影构建新的 SPJ 的投影。

- 在 query_planner 函数中通过 build_base_rel_tlists 函数获取投影中所需的所有的列，把在 SPJ 优化阶段将所有需要的列都进行投影，这对 Non-SPJ 阶段的投影优化是一个非常重要的保障，因为在 query_planner 之后基于 SPJ 的最优路径已经生成，这时如果在 Non-SPJ 优化阶段发现有些需要的投影列在 SPJ 优化阶段没有处理，这时候就会出错（或者说这时候再去改造已经产生的优化路径，添加新的投影列会增加无谓的代价）。

- 在 grouping_planner 函数中重新构造各个 Non-SPJ 阶段的投影列，根据 Non-SPJ 路径的特点，构建不同的投影。

因此在 Non-SPJ 优化阶段通过 make_window_input_target 函数和 make_group_input_target 函数来构建适用于 SPJ 阶段（scan/join）的投影（scanjoin_target），新的投影列要应用到 SPJ 优化阶段产生的路径上，这会带来如下问题：

- 一些路径不能直接进行投影，需要在这些路径上叠加一层 Result 路径，通过 Result 路径来进行投影，这就是 apply_projection_to_path 函数和 create_projection_path 函数的区别。
- 用新的路径替换原来的路径，路径的投影代价会发生变化，需要对代价的变化进行调整。
- 并行路径在应用这些投影时需要保证投影中的表达式是并行安全的，否则并行路径不能使用。

9.2.2.2　聚集与分组

如果分组条件中全部是空集合而且查询语句中没有聚集函数，那么这个语句是可以优化的，例如下面示例：

```
postgres=# EXPLAIN SELECT NEXTVAL('SEQ') FROM TEST_A GROUP BY GROUPING SETS((),());
                QUERY PLAN
-------------------------------------------
 Append  (cost=0.00..0.03 rows=2 width=8)
   ->  Result  (cost=0.00..0.01 rows=1 width=8)
   ->  Result  (cost=0.00..0.01 rows=1 width=8)
(3 rows)
```

分组路径的实现主要有两种手段。

基于排序实现的分组，是否可以使用排序分组需要满足如下条件之一：

- 在预处理 Group By 子句时，为能够适用于排序的 Group By 条件生成了基于 ROLLUP 的 Group Keys，目的是让多个分组共用一次排序结果。
- 每个 Group Keys 都可以满足基于排序的操作符（即有可以用来排序的操作符）。

基于 Hash 实现的分组，是否可以使用基于 Hash 分组需要满足如下条件：

- 查询中必须包含 Group By 子句才会生成基于 Hash 分组的路径（反过来不一定成立，例如 SELECT DISTINCT a FROM TEST_A 中没有 Group By 子句，仍然可能使用 Hash 分组）。
- 聚集函数中不能包含排序的情况，聚集函数中不能包含 DISTINCT，这些标记在

get_agg_clause_costs 函数中已经获得。

- 如果不是简单的 Group By 子句，而是带有 Grouping sets、ROLLUP、CUBE 的子句，会形成多组 Group Keys，主要部分 Group Keys 能够支持 Hash 分组，则可以生成基于 Hash 分组的路径。

目前 PostgreSQL 数据库支持并行聚集的实现，并行聚集需要满足如下条件：

- 路径中的表达式需要是并行安全的，例如投影中的表达式、Having 子句中的表达式等。
- 聚集的子路径也需要是并行路径，例如要通过并行聚集来实现 SELECT SUM(a) FROM TEST_A，则要求 TEST_A 的扫描路径中需要有并行扫描路径，也就是说并行聚集函数的实现还需要依赖“数据来源”的并行。
- 路径中或者有聚集函数、或者有 Group By 子句，但是 Group By 子句中不能包含 Grouping Sets、ROLLUP、CUDE。
- 在 get_agg_clause_costs 函数中是否已经确定了支持并行聚集的实现。

并行的聚集实现是分两个阶段实现的，第一个阶段各个并行的工作进程负责对部分数据进行聚集操作，这些数据交给 Gather 线程负责收集，这个阶段称为 Partial 聚集。第二个阶段是从 Gather 线程获取第一个阶段的数据，然后对数据进行二次聚集，获得最终结果，这个阶段称为 Finalize 聚集，例如：

```
postgres=# EXPLAIN VERBOSE SELECT SUM(a) FROM TEST_A;
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=7793.55..7793.56 rows=1 width=8)
   Output: sum(a)
   -> Gather  (cost=7793.33..7793.54 rows=2 width=8)
        Output: (PARTIAL sum(a))
        Workers Planned: 2
        -> Partial Aggregate  (cost=6793.33..6793.34 rows=1 width=8)
             Output: PARTIAL sum(a)
             -> Parallel Seq Scan on public.test_a
                  Output: a
(9 rows)
```

因此，对于并行的聚集函数需要创建两个聚集的节点和一个 Gather 节点，聚集路径的创建流程如图 9-4 所示。

严禁网络传播本PDF，违者责任自负！

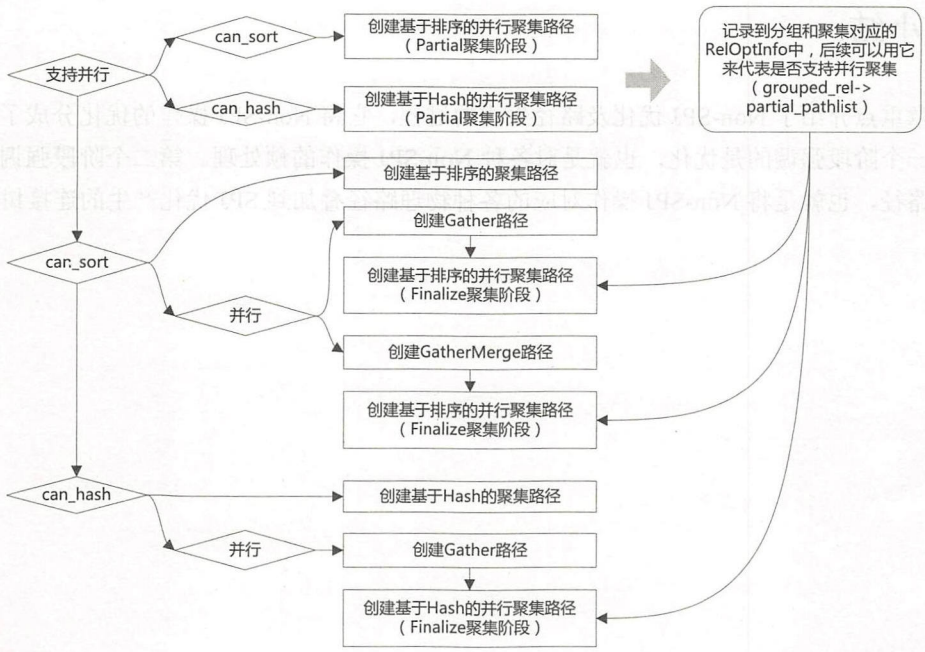


图 9-4　聚集路径生成流程

需要注意的是 Gather 节点和 GatherMerge 节点的区别，下面给出一个示例。

```
postgres=# EXPLAIN SELECT * FROM TEST_A;
                          QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=1000.00..270830.51 rows=28159660 width=16)
   Workers Planned: 2
   ->  Parallel Seq Scan on test_a  (cost=0.00..269548.92 rows=11733192 width=16)
(3 rows)
postgres=# EXPLAIN SELECT * FROM TEST_A ORDER BY a;;
                            QUERY PLAN
-------------------------------------------------------------------------------------
 Gather Merge  (cost=2049302.75..2323514.79 rows=23466384 width=16)
   Workers Planned: 2
   ->  Sort  (cost=2048302.73..2077635.71 rows=11733192 width=16)
         Sort Key: a
         ->  Parallel Seq Scan on test_a  (cost=0.00..269548.92 rows=11733192 width=16)
(5 rows)
```

从示例可以看出，GatherMerge 节点会保证收集到的数据的有序性。

严禁网络传播本PDF，违者责任自负！

9.3 小结

本章重点介绍了 Non-SPJ 优化及路径生成的过程，它将 Non-SPJ 操作的优化分成了两个阶段，第一个阶段强调的是优化，也就是对各种 Non-SPJ 操作的预处理。第二个阶段强调的是生成物理路径，也就是将 Non-SPJ 操作对应的各种物理路径叠加到 SPJ 优化产生的连接树上。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

10 第 10 章 生成执行计划

最优的执行路径生成后，虽然 Path Tree 已经足够清楚地指出查询计划要进行的物理操作，但是它的结构体中为了进行代价计算有太多的冗余信息，不方便查询执行器使用，并且有些参数还没有建立好，因此通过将其转换成执行计划来生成更适合查询执行器的 Plan Tree，然后将 Plan Tree 转交给执行器就可以真正执行了。路径的结构体是 Path 或者是“继承”自 Path 的新的路径结构体，例如 JoinPath，对应的执行计划节点的结构体是 Plan 或者是“继承”自 Plan 的新的执行计划节点，例如 Join 结构体。PostgreSQL 数据库的每个 Path 节点都一一对应一个 Plan 节点，最优的执行路径需要通过 create_plan 函数转换成对应的执行计划。

10.1 转换流程

路径（Path）可以分成扫描路径和连接路径，转换成执行计划的源代码中也做了这种区分，如果扫描路径转换成扫描的执行计划节点，用的是继承自 Plan 的 Scan 结构体，对应的连接路径则是 Join 结构体，所有的扫描计划节点都“继承”自 Scan 结构体，而所有的连接计划节点都“继承”自 Join 结构体，如图 10-1 所示。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

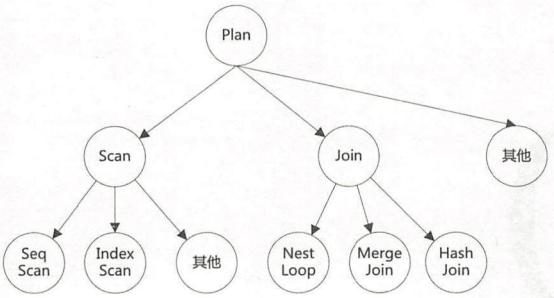


图 10-1　Plan 和 Path 的对应关系

在物理优化阶段已经选出了代价最低的最优路径，这里通过 create_plan 函数将其中的 Path 节点一一转换成为 Plan 节点，转换的过程如图 10-2 所示。

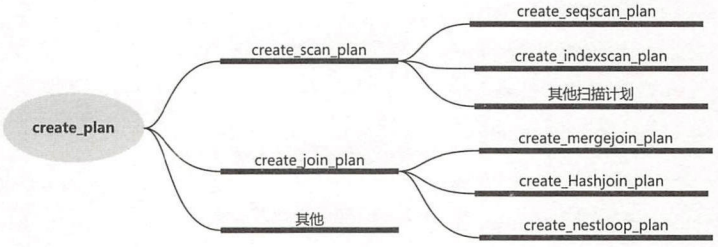


图 10-2　创建 Plan 的函数调用关系

10.1.1　扫描计划

在生成扫描路径时，我们着重介绍了 3 种扫描路径：顺序扫描路径、索引扫描路径、位图扫描路径。这些路径节点（Path）都要转换成新的计划节点（Plan），在分别转换每个路径之前，所有的扫描节点有一些公共的事情要处理，于是通过 create_scan_plan 函数先处理所有扫描路径转换过程中的公共部分。

通常而言，扫描节点上只有过滤条件，这些过滤条件可以通过 RelOptInfo->baserestrictinfo 来获得，但是索引扫描例外，索引结构体 IndexOptInfo 中通过 indrestrictinfo 也保存了一份过滤条件，indrestrictinfo 通常和 baserestrictinfo 相同，但是对于带有谓词的局部索引，indrestrictinfo 通常和 baserestrictinfo 可能是不同的，例如：

```
CREATE TABLE TEST_A(a INT, b INT, c INT, d INT);
CREATE INDEX TEST_A_A_GT2_IDX ON TEST_A(a) WHERE a >2;
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE a > 2;
                          QUERY PLAN
```


严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
-------------------------------------------------------------------------------------
 Index Scan using test_a_a_gt2_idx on test_a  (cost=0.29..328.27 rows=9999 width=16)
(1 row)
```

如果扫描节点的上层是连接操作，那么可能会有下推的参数化的约束条件，这些原本是连接型的过滤条件（连接型是指操作符两端都是变量的过滤条件）或者连接条件，由于操作符一端的变量可以参数化（可假设为常量），如果 join_clause_is_movable_to 函数判断其可以下推，那么这个约束条件就可以作为下层扫描节点的过滤条件，例如：

```
postgres=# EXPLAIN SELECT * FROM TEST_B,TEST_A WHERE TEST_B.b = TEST_A.b;
                          QUERY PLAN
-------------------------------------------------------------------------------------
 Nested Loop  (cost=0.29..30570.00 rows=999493 width=32)
   ->  Seq Scan on test_b  (cost=0.00..155.00 rows=10000 width=16)
   ->  Index Scan using test_a_b_idx on test_a  (cost=0.29..2.05 rows=99 width=16)
         Index Cond: (b = test_b.b)
(4 rows)
```

普通的约束条件和参数化的约束条件组合在一起，就形成了这个扫描节点上所有的“扫描条件（scan_cluase）”。

如果扫描条件中有常量，那么可以把这个常量的扫描条件找出来，作为 gating 约束条件，因为常量的约束条件不受执行过程中变量的影响，因此可能带来一些优化。

```
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE EXISTS (SELECT 1 FROM TEST_B WHERE A < 0) AND
EXISTS (SELECT 1 FROM TEST_C WHERE B < 0);
                        QUERY PLAN
-----------------------------------------------------------------------
 Result  (cost=360.00..515.00 rows=10000 width=16)
   One-Time Filter: ($0 AND $1)
   InitPlan 1 (returns $0)
     ->  Seq Scan on test_b  (cost=0.00..180.00 rows=1 width=0)
           Filter: (a < 0)
   InitPlan 2 (returns $1)
     ->  Seq Scan on test_c  (cost=0.00..180.00 rows=1 width=0)
           Filter: (b < 0)
   ->  Seq Scan on test_a  (cost=360.00..515.00 rows=10000 width=16)
(9 rows)
```

另外对投影列也可以适当地做一些优化，虽然我们只打算对一个表的某些列做投影，但是如果这些表是可以扫描的基表，那么可以把它的所有列都投影出来，这样查询执行时可以提高效率，例如下面的示例：

非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
postgres=# EXPLAIN VERBOSE SELECT TEST_A.a,TEST_B.b FROM TEST_A, TEST_B;
                          QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..1250335.00 rows=100000000 width=8)
   Output: test_a.a, test_b.b
   ->  Seq Scan on public.test_a  (cost=0.00..155.00 rows=10000 width=4)
         Output: test_a.a, test_a.b, test_a.c, test_a.d
   ->  Materialize  (cost=0.00..205.00 rows=10000 width=4)
         Output: test_b.b
         ->  Seq Scan on public.test_b  (cost=0.00..155.00 rows=10000 width=4)
               Output: test_b.b
(8 rows)
```

在示例的 SQL 语句中，TEST_A 表中只有 a 列需要进行投影，但是在对 TEST_A 表进行顺序扫描时，对表中的所有列都做了投影，和 TEST_A 表不同，SQL 语句中 TEST_B 表没有对整个表的列进行投影，而是在 SeqScan 的时候只投影了 b 列，这是因为 TEST_B 的 SeqScan 路径的上层节点是物化路径，物化路径要求保持最小的投影结果（CP_SMALL_TLIST），在 create_plan 行数遍历最优路径中的各个节点的过程中，如果发现诸如 Hash、Sort、Materia 之类的节点，会设置 CP_SMALL_TLIST 标志，这样下层路径就不会随便扩展投影列了，因为这时候扩展投影列会带来无谓的代价消耗。

```
static Material *
create_material_plan(PlannerInfo *root, MaterialPath *best_path, int flags)
{
   subplan = create_plan_recurse(root, best_path->subpath,
                                 flags | CP_SMALL_TLIST);

   plan = make_material(subplan);
   copy_generic_path_info(&plan->plan, (Path *) best_path);
   return plan;
}
```

create_scan_plan 函数处理了所有扫描路径公共的特性之后，开始单独处理每个扫描路径向扫描计划的转换。

10.1.1.1 顺序执行计划

这时候我们已经获得了扫描的约束条件和扫描产生的投影列，对于顺序扫描而言已经万事俱备，只需要对顺序扫描路径中的数据做一些调整，就可以生成扫描执行计划。

首先，调整约束条件的执行顺序，这里调整的规则涉及了安全级别和表达式的执行代价（本

非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

书不打算介绍安全级别相关的知识），在安全级别相同的情况下，以约束条件中的表达式代价作为排序的依据，表达式代价越低的约束条件排序越靠前。

其次，在查询优化过程中，所有的约束条件都是使用 RestrictInfo 结构体来表示的，这个结构体主要的作用是记录和约束条件相关的物理优化信息（例如等价类信息、代价信息等），这些信息在创建和筛选 Path 的过程中都发挥了作用，但是到了生成执行计划的阶段，这些信息已经变成了冗余信息，在查询执行器中只需要知道待执行的表达式就可以了，因此还需要从 RestrictInfo 中将表达式提取出来，查询执行器只需要执行这些表达式就可以了。

再次，如果当前扫描路径是参数化路径，那么需要检查约束条件中是否引用了外表的列作为参数，如果引用了外表的列（Var 或 PlaceHolderVar），那么把它们替换成 Param，并且建立对应的 NestloopParam 结构体。目前的 PostgreSQL 数据库在顺序扫描的约束条件中不会出现参数化路径的情况，它的参数主要出现在投影列中，但是投影列的参数替换是在 create_scan_plan->build_path_tlist 函数中进行的。

10.1.1.2　索引执行计划

索引扫描执行计划的建立过程要稍复杂一些，因为索引扫描涉及一些约束条件的处理，例如索引扫描将约束条件分成了两类，分别是能够匹配索引的约束条件和不能匹配索引的约束条件。

能够匹配索引的约束条件中的 Var 使用的还是索引所在表上的 Var，为了方便查询执行器进行索引扫描需要将这种 Var 替换成为索引上的 Var，例如对于表 TEST_A(a,b,c,d)和表上的索引 TEST_A_B_IDX(b)，约束条件 b=1 目前是通过 TEST_A(b)来标识约束条件中的 b 属性的，但是在索引扫描执行时，实际上需要 TEST_A_B_IDX(b)来标识，这样查询执行器才能知道使用索引的哪一个属性来应用约束条件，这种替换操作是通过 fix_indexqual_references 函数来实现的，在创建索引扫描路径时，对约束条件和索引匹配主要针对的是 Var、OpExpr、ScalarArrayOpExpr、RowCompareExpr、NullTest 等几种表达式，在 fix_indexqual_references 函数中也主要处理了这几种表达式，而替换的操作则是在 fix_indexqual_operand 函数中实现的，另外 fix_indexqual_operand 函数还处理了表达式索引的情况。

```
//确定是表上的Var
if (IsA(node, Var) &&
   ((Var *) node)->varno == index->rel->relid &&
   ((Var *) node)->varattno == index->indexkeys[indexcol])
{
   //生成基于索引的Var
   result = (Var *) copyObject(node);
```


非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
    result->varno = INDEX_VAR;
    result->varattno = indexcol + 1;
    return (Node *) result;
}
```

不能匹配索引的约束条件则作为过滤条件对索引扫描的结果进行过滤，例如：

```
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE a=1 and b > 2;
                          QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using test_a_a_idx on test_a  (cost=0.29..8.30 rows=1 width=16)
   Index Cond: (a = 1)
   Filter: (b > 2)
(3 rows)
```

示例 SQL 语句中的 a=1 能够匹配索引 TEST_A_A_IDX，因此是匹配索引的约束条件，而 b>2 则不能匹配索引，它在索引扫描之后进行二次过滤（Filter）。

目前，scan_clauses 中包含了所有的约束条件，而在 indexquals 中则只包含了和索引匹配的约束条件，因此用 scan_clauses 减去 indexquals 获得的就是所有未匹配索引的过滤条件，这个“减法”操作的实现如下：

```
foreach(l, scan_clauses)
{
    RestrictInfo *rinfo = lfirst_node(RestrictInfo, l);

    //常量约束条件在create_scan_plan函数中已经挑选出来了，这里跳过不处理
    if (rinfo->pseudoconstant)
        continue;           /* we may drop pseudoconstants here */
    //约束条件如果是能匹配索引的约束条件，那么跳过不处理
    if (list_member_ptr(indexquals, rinfo))
        continue;           /* simple duplicate */

    //如果约束条件和匹配索引的条件是基于同一个等价类的
    //也就是说这个约束条件是多余的，那么这里跳过不处理
    if (is_redundant_derived_clause(rinfo, indexquals))
        continue;           /* derived from same EquivalenceClass */

    //如果索引约束条件蕴含一个约束条件，那么这个约束条件也跳过不处理
    if (!contain_mutable_functions((Node *) rinfo->clause) &&
        predicate_implied_by(list_make1(rinfo->clause), indexquals, false))
        continue;           /* provably implied by indexquals */
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
    //收集未匹配索引的过滤条件
    qpqual = lappend(qpqual, rinfo);
}
```

predicate_implied_by 函数用来判断约束条件之间的蕴含关系，如果把 a=1 这样单独的约束条件称为 atom，把(a=1 OR c=2) 这样的约束条件称为 OR-expr ，把(a=1 AND c=2)这样的约束条件称为 AND-expr，那么 PostgreSQL 数据库可以支持的蕴含关系如下：

```
atom A => atom B iff:              通过predicate_implied_by_simple_clause函数验证
atom A => AND-expr B iff:          A => each of B's components
atom A => OR-expr B iff:           A => any of B's components
AND-expr A => atom B iff:          any of A's components => B
AND-expr A => AND-expr B iff:      A => each of B's components
AND-expr A => OR-expr B iff:       A => any of B's components,
                                   *or* any of A's components => B
OR-expr A => atom B iff:           each of A's components => B
OR-expr A => AND-expr B iff:       A => each of B's components
OR-expr A => OR-expr B iff:        each of A's components => any of B's
```

predicate_implied_by_recurse 函数对上面的 9 种情况分别做了判断，从下面的示例可以看出，执行计划中没有 c=2 这个约束条件了。

```
-- atom A => OR-expr B
-- SQL语句中的a=1如果成立，那么(a=1 OR c=2)也一定成立
-- 那么它们之间就是(a=1)  =>  (a=1 or c=2)的关系
postgres=# EXPLAIN SELECT * FROM TEST_A WHERE a=1 AND (a=1 OR c=2);
                          QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using test_a_ab_idx on test_a  (cost=0.29..8.31 rows=1 width=16)
   Index Cond: (a = 1)
(2 rows)
```

索引扫描计划的生成还需要对约束条件进行“排序”和“提取”，这些操作和顺序扫描计划生成时类似，这里不再赘述。需要注意的是对 indexorderbys 的处理，在创建索引扫描路径时我们已经说过目前只有 Gist 索引支持这种特殊情况，当时已经给出了示例，这里不妨把示例再展示出来，有兴趣的读者可自行分析。

```
CREATE TABLE POINT_TBL(f1 POINT);
CREATE INDEX gpointind ON POINT_TBL USING gist (f1);
EXPLAIN SELECT * FROM point_tbl ORDER BY f1 <-> '0,1';
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

10.1.2 连接计划

连接路径节点转换成连接执行计划节点分成了 MergeJoin、HashJoin、NestloopJoin 三种情况，大部分的处理和扫描节点大同小异，我们选择以 MergeJoin 为例做一些简单的说明。

连接路径节点首先要递归调用 create_plan_recurse 函数将子路径替换成子计划，子计划如果是基表扫描节点，它的投影列要不要扩展取决于连接节点是否需要显式地排序，如果子路径需要显式地排序，那么就指定 CP_SMALL_TLIST 标志给子计划，这样就可以避免对无用的数据进行排序。

连接路径的约束条件可以分成连接条件和过滤条件，如果连接（Join）不是外连接，那么实际上连接条件和过滤条件的区别不大，例如：

```
postgres=# EXPLAIN SELECT * FROM TEST_A INNER JOIN TEST_B ON TEST_A.a = TEST_B.a WHERE TEST_A.b
= TEST_B.b;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
-
 Merge Join  (cost=819.68..1344.54 rows=100 width=32)
   Merge Cond: ((test_a.a = test_b.a) AND (test_a.b = test_b.b))
   ->  Index Scan using test_a_ab_idx on test_a  (cost=0.29..399.16 rows=10000 width=16)
   ->  Sort  (cost=819.39..844.39 rows=10000 width=16)
         Sort Key: test_b.a, test_b.b
         ->  Seq Scan on test_b  (cost=0.00..155.00 rows=10000 width=16)
(6 rows)
```

SQL 语句中的 TEST_A.a = TEST_B.a 是连接条件，而 TEST_A.b = TEST_B.b 是过滤条件，但由于是内连接，它们没有什么区别，但是对于外连接则不同，由于外连接需要补 NULL 值，要对连接条件和过滤条件进行区分，在 RestrictInfo 结构体中提供了 is_pushed_down 变量来进行区分，is_pushed_down = true 代表这个约束条件是过滤条件。

连接条件中又可以分成 MergeJoinable 的连接条件和其他连接条件，也需要进行区分，例如：

```
postgres=# EXPLAIN SELECT * FROM TEST_A LEFT JOIN TEST_B ON TEST_A.a = TEST_B.a AND TEST_A.b
= 1;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Merge Left Join  (cost=819.68..1347.67 rows=10000 width=32)
   Merge Cond: (test_a.a = test_b.a)
   Join Filter: (test_a.b = 1)
   ->  Index Scan using test_a_a_idx on test_a  (cost=0.29..328.29 rows=10000 width=16)
```


非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
  ->  Sort  (cost=819.39..844.39 rows=10000 width=16)
        Sort Key: test_b.a
        ->  Seq Scan on test_b  (cost=0.00..155.00 rows=10000 width=16)
(7 rows)
```

示例 SQL 语句的中 TEST_A.a = TEST_B.a 是 MergeJoinable 的连接条件，而 TEST_A.b=1 就是普通的连接条件。

MergeJoin 还可能需要对两个子计划进行显式的排序，这时候就需要我们根据 MergeJoin 路径中的 PathKeys 增加对应的 Sort Plan（注意：虽然这个 SortPlan 现在才刚刚生成，但是 MergeJoin 计算路径代价的时候已经显式地计入了 Sort 的代价），如图 10-3 所示。

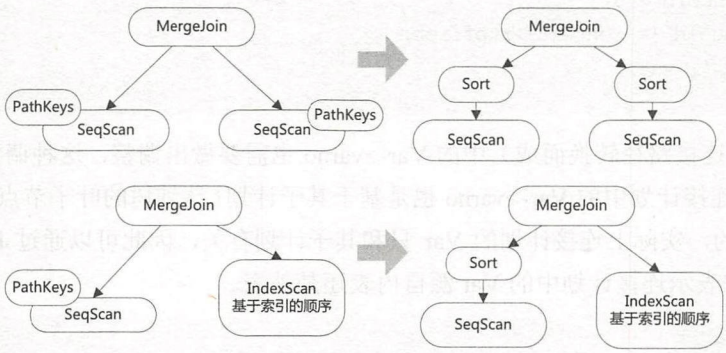


图 10-3　MergeJoin 增加 Sort 节点

10.2　执行计划树清理

计划树已经生成，目前要做的是查询优化器的最后一步，对执行计划树进行最后的整理，最主要的工作是将子计划的范围表拉平到和父计划的范围表放到一起，如图 10-4 所示。

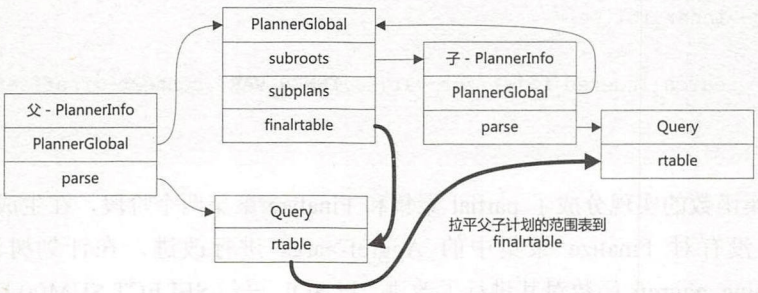


图 10-4　计划树拉平

非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！

范围表拉平之后，计划树中扫描计划的 Var 变量的 varno 也会发生变化，因此还需要遍历计划树中的 Var 变量，调整其中的 varno 的值（fix_scan_list 函数）。

```
//fix_scan_expr_mutator 函数，与其他表达式类似，都需要将 varno 增加对应的偏移量
if (IsA(node, Var))
{
   Var         *var = copyVar((Var *) node);

   //子计划中的范围表的位置发生了变化，给它增加相应的偏移量
   if (!IS_SPECIAL_VARNO(var->varno))
      var->varno += context->rtoffset;
   if (var->varnoold > 0)
      var->varnoold += context->rtoffset;
   return (Node *) var;
}
```

连接计划（连接路径转换而成）中的 Var->varno 也需要做出调整，这种调整和扫描计划的调整不同，目前连接计划中的 Var->varno 也是基于其子计划（计划树的叶子节点，即扫描计划）的 rtindex 而来的，实际上连接计划的 Var 只和其子计划有关，因此可以通过 INNER_VAR 和 OUTER_VAR 来表示连接计划中的 Var 源自内表还是外表。

```
if (IsA(node, Var))
{
   Var  *var = (Var *) node;

   //如果源自外表的 Var
   if (context->outer_itlist)
   {
      newvar = search_indexed_tlist_for_var(……,OUTER_VAR, context->rtoffset);
   }
   //如果源自内表的 Var
   if (context->inner_itlist)
   {
      newvar = search_indexed_tlist_for_var(……,INNER_VAR, context->rtoffset);
   }
}
```

并行的聚集函数的实现分成了 partial 聚集和 Finalize 聚集两个阶段，在生成执行路径和执行计划阶段，没有对 Finalize 聚集中的 Aggref->args 进行改进，在计划树清理阶段通过 convert_combining_aggrefs 函数对其进行了改进。以 SQL 语句 SELECT SUM(a) FROM TEST_A 为例，如图 10-5 所示，左图是 SPJ 优化之后的情况，Partial 聚集和 Finalize 聚集都是使用的同

非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

一个 Var，现在对这个 Var 进行修正。

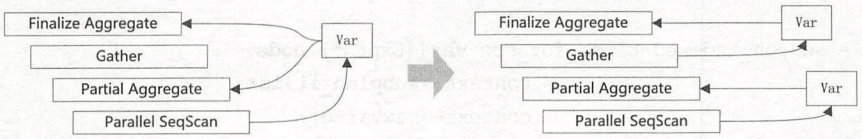


图 10-5　并行聚集函数的修正

```
//用原始的 Aggref 生成新的 parent_agg 和 child_agg
//child_agg 作为 parent_agg 的参数，注意这里修改的是 combining 聚集
if (IsA(node, Aggref))
{
    //生成 child_agg
    child_agg = makeNode(Aggref);
    memcpy(child_agg, orig_agg, sizeof(Aggref));

    //生成 parent_agg
    child_agg->args = NIL;
    child_agg->aggfilter = NULL;
    parent_agg = copyObject(child_agg);
    child_agg->args = orig_agg->args;
    child_agg->aggfilter = orig_agg->aggfilter;

    /*
     * Now, set up child_agg to represent the first phase of partial
     * aggregation.  For now, assume serialization is required.
     */
    mark_partial_aggref(child_agg, AGGSPLIT_INITIAL_SERIAL);

    //child_agg 作为 parent_agg 的参数
    parent_agg->args = list_make1(makeTargetEntry((Expr *) child_agg,
                                    1, NULL, false));
    mark_partial_aggref(parent_agg, AGGSPLIT_FINAL_DESERIAL);

    return (Node *) parent_agg;
}
```

在 set_plan_refs 函数中，对 Finalize 聚集修改之后会进入 set_upper_references，它会将 child_agg 替换成基于 gather 计划的投影的 Var。

```
//subplan_itlist 是 gather 计划的投影
```


非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
//node 是 child_agg 表达式
if (context->subplan_itlist->has_non_vars)
{
   newvar = search_indexed_tlist_for_non_var((Expr *) node,
                                    context->subplan_itlist,
                                    context->newvarno);
   if (newvar)
      return (Node *) newvar;
}
```

PARAM_MULTIEXPR 类型的参数也需要在这里进行改进，如下示例中，PARAM_MULTIEXPR 被改进成多个 PARAM_EXEC 类型的参数（由 PARAM_MULTIEXPR 向 PARAM_EXEC 的转换在 generate_subquery_params 函数中实现）。

```
postgres=# EXPLAIN VERBOSE UPDATE TEST_A SET (a, b, c, d) = (SELECT a, b, c, d FROM TEST_B);
                          QUERY PLAN
------------------------------------------------------------------------------
 Update on public.test_a  (cost=155.00..364.00 rows=10000 width=54)
   InitPlan 1 (returns $0,$1,$2,$3)
     -> Seq Scan on public.test_b  (cost=0.00..155.00 rows=10000 width=16)
          Output: test_b.a, test_b.b, test_b.c, test_b.d
   -> Seq Scan on public.test_a  (cost=0.00..209.00 rows=10000 width=54)
        Output: $0, $1, $2, $3, NULL::record, test_a.ctid
(6 rows)
```

清理阶段还会通过 record_plan_function_dependency 记录用户自定义函数与执行计划的依赖关系，这个依赖关系提供给 PostgreSQL 的缓存模块，如果缓存模块有新的失效消息，则执行计划也需要失效。

最后的一项优化是 SubqueryScan 的消除，如果符合以下条件：

- SubqueryScan 计划中没有约束条件。
- SubqueryScan 的投影链表和子计划的投影链表相同。

这意味着当前的 SubqueryScan 不需要做什么实际的工作，只是对子计划的再次投影，因此可以将其消除掉。例如 SQL 语句 SELECT * FROM TEST_A LEFT JOIN (SELECT * FROM TEST_B LIMIT 10) b ON TRUE，它原本的查询计划应该如下：

```
NestloopJoin - LeftJoin
     -> SeqScan - TEST_A
     ->Material
               ->SubqueryScan
```


非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
                    ->Limit 10
                        ->SeqScan - TEST_B
```

但由于 SubqueryScan 和 Limit 子句的投影相同，而且 SubqueryScan 本身没有约束条件，因此可以消除掉，实际的执行计划如下：

```
postgres=# EXPLAIN SELECT * FROM TEST_A LEFT JOIN (SELECT * FROM TEST_B LIMIT 10) b ON TRUE;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..1459.28 rows=100000 width=32)
   ->  Seq Scan on test_a  (cost=0.00..209.00 rows=10000 width=16)
   ->  Materialize  (cost=0.00..0.30 rows=10 width=16)
         ->  Limit  (cost=0.00..0.16 rows=10 width=16)
               ->  Seq Scan on test_b  (cost=0.00..155.00 rows=10000 width=16)
(5 rows)
```

10.3　小结

物理路径最终需要转换成物理执行计划，本章重点介绍了顺序扫描路径、索引扫描路径、MergeJoin 连接路径的转换过程，这些示例虽然具有一定的代表性，但是有兴趣的读者不妨分析一下其他类型的路径向执行计划转换的流程。